МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
КРИВОРІЗЬКИЙ ДЕРЖАВНИЙ ПЕДАГОГІЧНИЙ УНІВЕРСИТЕТ
Фізико-математичний факультет
Кафедра інформатики та прикладної математики

«Допущено до захисту»
В.о. завідувача кафедри
_____ Семеріков С.О.
«_____» _____ 2025 р.

Реєстраційний № _____
«_____» _____ 2025 р.

Розробка гри у жанрі Roguelike засобами рушія Godot Engine

Кваліфікаційна робота студента
групи І-21
ступінь вищої освіти «бакалавр»
спеціальності 014 Середня освіта
(Інформатика)
Бойченка Максима Андрійовича

Керівник: доц., к. ф.-м.н.
Тарасова Олена Юріївна

Оцінка:
Національна шкала_____
Шкала ECTS _____
Кількість балів _____
Голова ЕК _____ _____
Члени ЕК _____ _____
_____ _____
_____ _____

Кривий Ріг – 2025

ЗАПЕВНЕННЯ

Я, Бойченко Максим Андрійович, розумію і підтримую політику Криворізького державного педагогічного університету з академічної доброчесності. Запевняю, що ця кваліфікаційна робота виконана самостійно, не містить академічного плагіату, фабрикації, фальсифікації. Я не надавав і не одержував недозволену допомогу під час підготовки цієї роботи. Використання ідей, результатів і текстів інших авторів мають покликання на відповідне джерело.

Із чинним Положенням про запобігання та виявлення академічного плагіату в роботах здобувачів вищої освіти Криворізького державного педагогічного університету ознайомлений. Чітко усвідомлюю, що в разі виявлення у кваліфікаційній роботі порушення академічної доброчесності робота не допускається до захисту або оцінюється незадовільно.

ЗМІСТ

ВСТУП

У сучасному світі комп’ютерні ігри відіграють важливу роль у повсякденному дозвіллі. Як правило, їх використовують для того, щоб розважити себе та відволіктись від складнощів та турбот. За увесь час розвитку, ігри перейшли від маленьких текстових проєктів, які породили більшу кількість жанрів, до великих ігор з графікою, об’ємними моделями, сюжетом, механіками тощо.

Всесвітній ринок ігрової індустрії постійно зростає, приваблюючи все більше інвестицій та ентузіастів у проєкти. Не обов’язково бути великою компанією зі штатом співробітників, щоб створити гру мрії. Останнім часом популярність здобули indie проєкти, у яких розробники постійно експериментують над створенням нових механік ігрового процесу, не переживаючи про наслідки.

Серед усіх подібних проєктів, нас привабив жанр Roguelike. Безмежний потенціал генерації нових рівнів дозволить користувачу постійно отримувати новий досвід під час проведення часу у грі. Такі ігри можуть розвивати у людини реакцію, дрібну моторику, логічне та критичне мислення.

Більшість indie проєктів написані за допомогою рушія Unity, проте через введення нових правил політики даного рушія, він перестав бути повністю безкоштовним, через що очевидним став вибір рушія GoDot Engine, як швидкого та простого аналога.

Саме через це, була обрана тема дипломної роботи «Розробка гри у жанрі roguelike засобами рушія Godot Engine», тому що це можливість дізнатись більше про історію зародження та розвитку ігор жанру roguelike, а також проаналізувати конкурента рушія Unity для створення ігор.

Мета цієї дипломної роботи – розробити гру в жанрі Roguelike, використовуючи функціональні можливості ігрового рушія Godot Engine, на основі дослідження характеристик жанру та інструментарію рушія.

Об’єктом дослідження є принципи геймдизайну та технологічні аспекти створення комп'ютерних ігор.

Предметом дослідження є методологія розробки гри жанру roguelike з використанням ігрового рушія Godot Engine.

Відповідно до мети, об’єкту та предмету було визначено наступні завдання роботи:

1. проаналізувати історію зародження та розвитку жанру roguelike, дослідити ігри цього жанру;
2. визначити характерні риси жанру;
3. розібрати інструментарій ігрового рушія Godot Engine;
4. проаналізувати можливості скриптової мови GDScript;
5. створити власну ігрову розробку.

Структура та обсяг: дипломна робота складається із вступу, чотирьох розділів, висновків до кожного розділу та загальних висновків. Загальний обсяг роботи – 74 сторінок, з них 66 основного тексту. Список використаних джерел на 3 сторінках містить 26 найменувань.

РОЗДІЛ 1
АНАЛІЗ ОСОБЛИВОСТЕЙ ІГОР ЖАНРУ ROGUELIKE

1.1. Історія жанру

Roguelike – це жанр комп'ютерних ігор, що характеризується великою кількістю випадкових подій, складності ігрового процесу, процедурної генерації рівнів та перманентною смертю персонажа [22]. Дослівного перекладу цієї назви не існує. Загалом, вона означає буквально «ігри, на кшталт Rogue». Як раз до цього перекладу підходить назва першої гри у цьому жанрі – «Rogue», яка вийшла у світ в 1980 році та була розроблена програмістом Гленном Вічмандом [12]. Гра була призначена для UNIX-подібних операційних систем. Ігровий процес представляє собою вивчення випадково згенерованого підземелля, у якому гравець повинен відбиватись від монстрів, та постійно знаходити амулет, щоб перейти на наступні рівні. Інтерфейс цієї гри представляє собою мапу, на якій позначається персонаж та вороги. Для користувача доступні характеристики персонажа, його рівень та кількість досвіду з ігровою валютою (рис. 1.1).

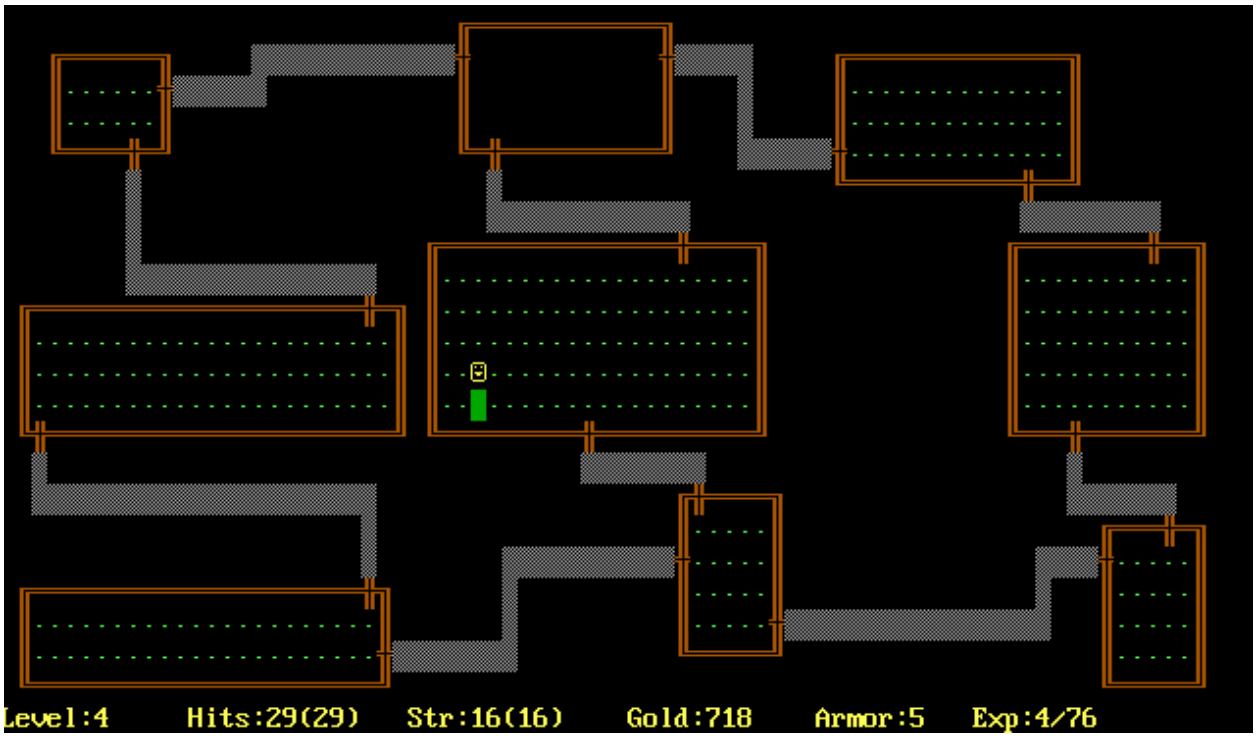

Рис. 1.1. Гра «Rogue»

Гра приваблювала користувачів свою ідеєю постійної генерації нових

локацій та варіантів підземелля. За допомогою цього, гра отримала ідею постійного перегравання та відсутністю кінцевої мети у грі.

Двома роками раніше, на платформі для персонального комп'ютера Apple II, було випущено гру під назвою «Beneath Apple Manor». Концептуально, ігри були дуже схожими. Процедурна генерація, пошук предметів у підземеллі були присутні у обох проєктах, проте розробники цих ігор затвердили, о не знали про розробку ігор одне одного.

Незважаючи на це, спільнота вважає гру «Rogue» прародителем жанру Roguelike. На основі цієї гри були реалізовані ігри з аналогічними механіками, такі як: «Hack», «NetHack» та «Moria» 1985-1987 років [20].

З усіх перелічених вище проєктів, найбільшу популярність здобула гра «NetHack» (рис. 1.2). Популярність вона здобула за допомогою своєї RPG складової та великої кількості пасток, які зустрічає гравець по ходу проходження лабіринту [1]. Гру створив студент Джей Фенлансон. Він хотів пограти у «Rogue», проте не мав доступу до неї. Саме через це, він вирішив створити власний гри по пам'яті. Через деякий час, його одногрупники допомогли йому у створення гри, через що гра, яка мала бути клоном «Rogue», перетворилась у самостійний проєкт, натхненний оригіналом [24].

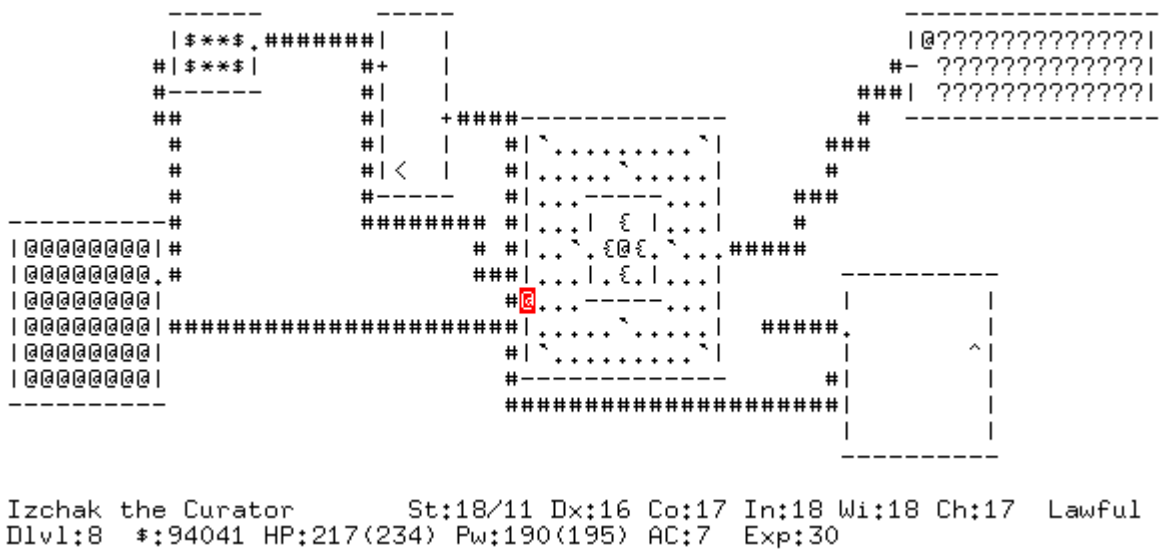

Рис. 1.2. Гра «NetHack»

Ігри активно оновлювались та переходили з системи на систему. Навіть

у моменти, коли розробники залишали свої проєкти, ентузіасти брали до їх оновлення та удосконалення. Але початковий розвиток не обмежувався лише цими іграми.

Фінський програміст Петрі Ніський у 1987 році створив свою інтерпретацію roguelike ігор. Замість популярних у своєму жанрі підземелля, лабіринтів та монстрів, у грі представлені вулиці, активні будинки, люди та комічна взаємодія з ними. Гра полюбилась користувачам, тому що вона була унікальною за ідеєю, а саме реалістичними подіями, а також зрозумілою реалізацією ігрового процесу. Кожна літера на екрані, представляла собою певного ігрового персонажа, з яким головному герою потрібно взаємодіяти, щоб знайти пляшку спиртного та перейти до наступного рівня [15].

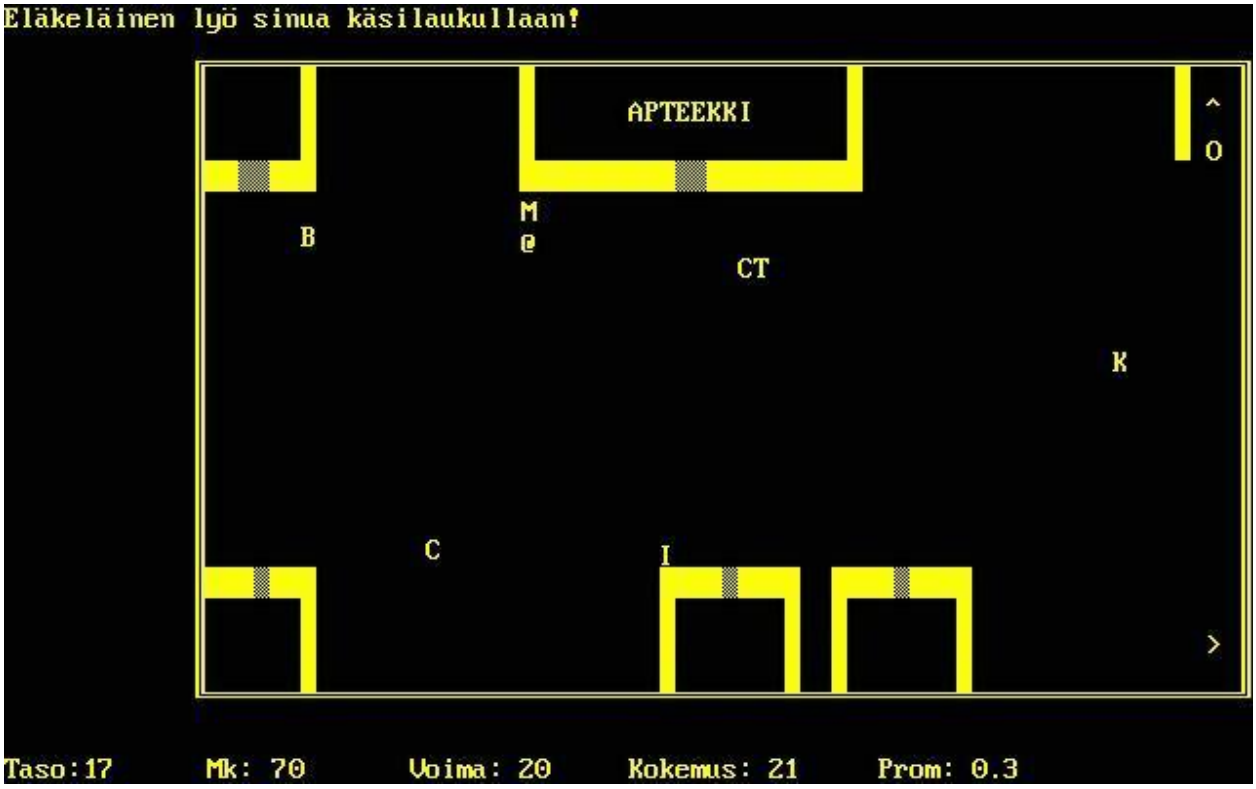

Рис. 1.3. Гра «SpurguX»

Прагнучи об’єднати усі особливості roguelike ігор того часу, програміст Ной Морган почав розробку особливого проєкта для жанру, а саме – «Larn». Починаючи з 1986 року, автор постійно додає нові механіки з ігор, кращих представників жанру. Таким чином, у грі присутні випадкова генерація схем підземелля, місто на поверхні для взаємодії з персонажами та удосконаленням, складність, яка залежала від глибини підземелля, таймер обмеження часу, для

спонукання гравця до активної гри, кількома варіаціями підземелля та інше [24].

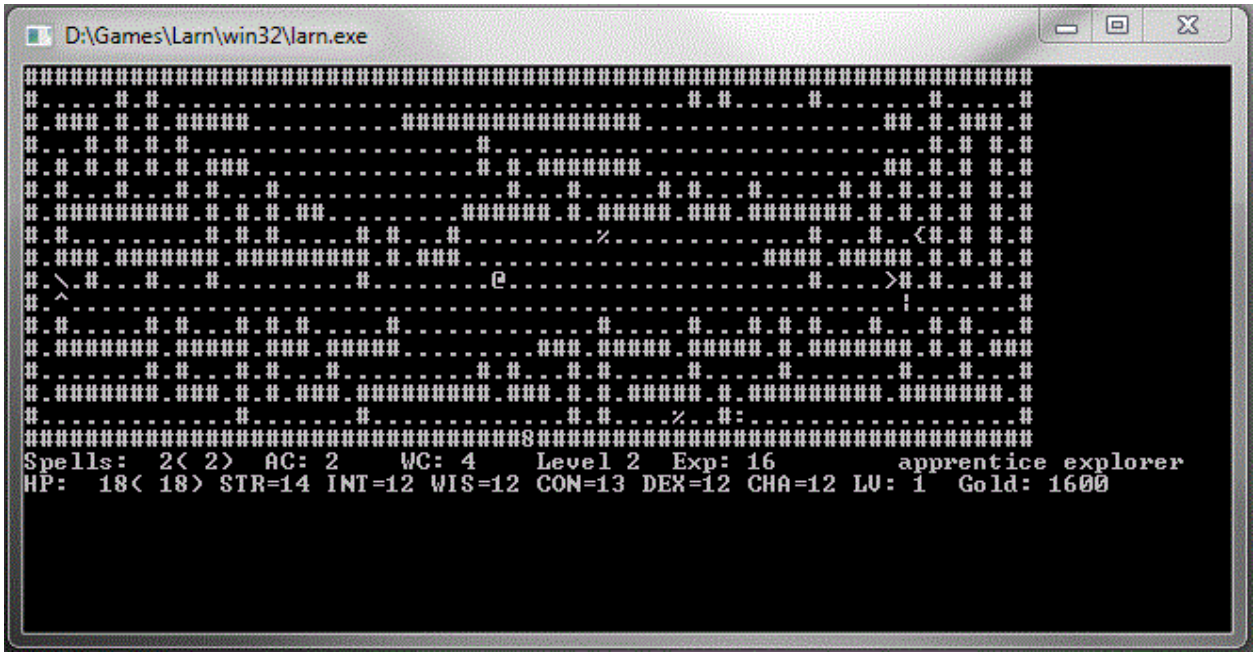

Рис. 1.4. Гра «Larn»

Наступні проєкти, такі як: «Umoria», «Angband», «Omega», не приносили нових механік, а лише удосконалювали готові ідеї, та повторювали одне одного. Усі ці ігри поєднує тільки те, що кожна з них ставала менш комфортною для гравця через те, що складність ставала дедалі більшою, а ігрові сесії довші. Ці проєкти допрацювали основні правила жанру та були останніми, перед ерою додавання графічної складової у гру.

У 1992 році, була випущена гра «UnReal World». Ключовою особливістю було те, що гра мала 2D графіку та позиціонувала себе, як виживання з елементами roguelike (рис. 1.5). Розробник Самі Мааранен підтримував гру оновленнями аж до 2009 року [22]. За свої інновації у поєднанні жанрів, у 2019 році, гра потрапила у книгу рекордів Гіннеса, як «Перша відеогра про виживання у відкритому світі» [3].

Ця гра дала поштовх до удосконалення графічної складової у вже існуючих іграх, та проєктах, які будуть виникати у майбутньому. Але не всі притримуватись такої думки, адже розробники хотіли зберігати ту атмосферу, яка була закладена при першій розробці гри.

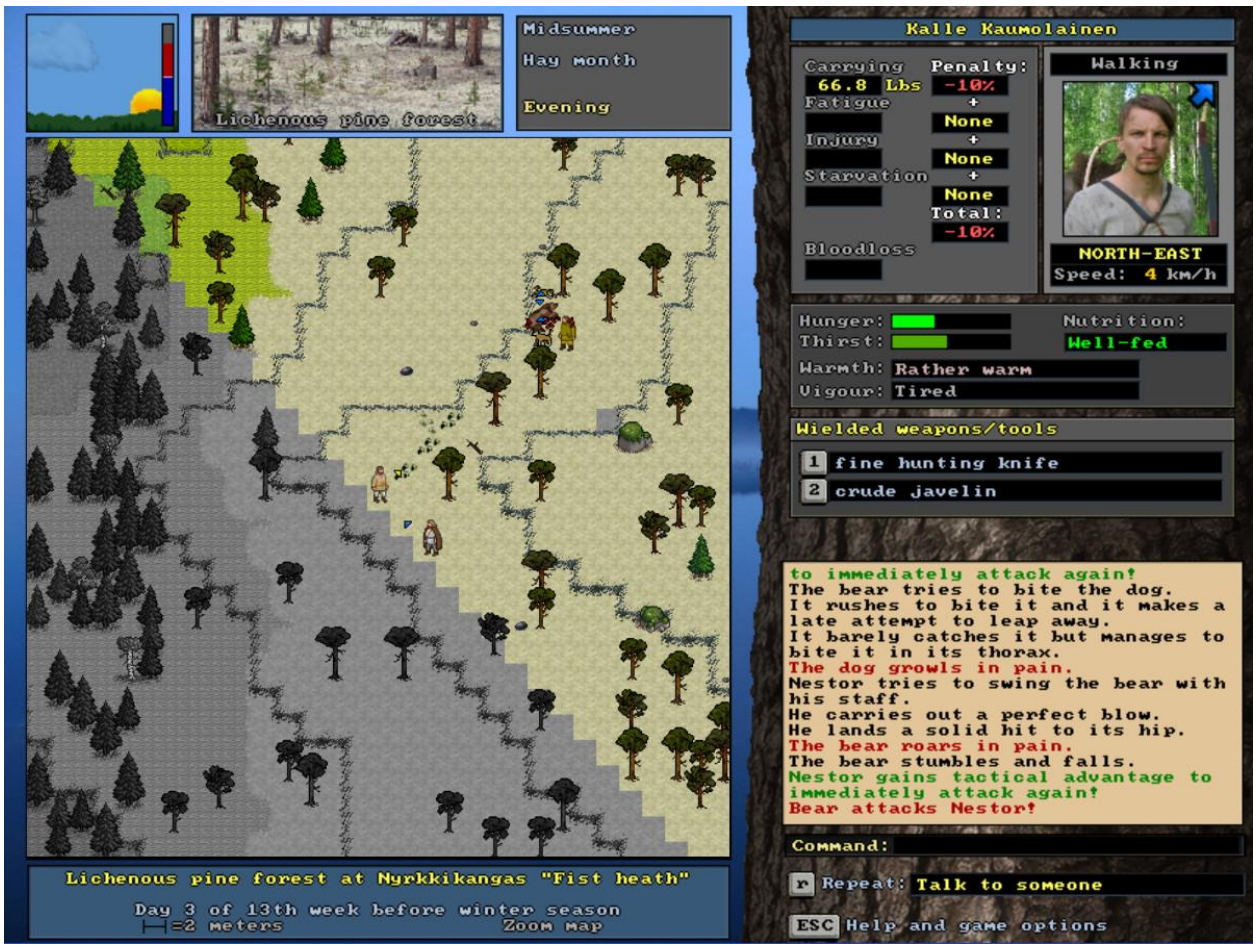

Рис. 1.5. Гра «UnReal World»

Саме так, гра «Omega», продовжувала експериментувати та вводити нові механіки для ускладнення або урізноманітнення ігрового процесу. Так з’явились механіки створення пасток, закритих дверей, тобто унеможливлення вийти з рівня, без повноцінної зачистки рівня, експерименти з тривалістю ігрових сесій та урізноманітнення варіантів взаємодії з ігровим світом [24].

З часом, у жанр roguelike почали додавати сюжет та прикріпляти міфологію до стилістики ігор. Наприклад проєкти Ragnarok та Valhalla активно описували скандинавські мотиви. Також додались класи, трансформація персонажа та квести.

Вихід жанру на японський ринок трохи затримався, але вже у 1986 році вийшла перша локалізація «Rogue» на японську мову. До 1990 року, на ринку комп’ютерних ігор Японії виникали лише копії та клони оригінальної гри. Вони не мали певних особливостей, але вже дозволяли людям експериментувати [26].

Так, у 1990 році вийшла перша масштабна гра у жанрі roguelike для консолі Sega Genesis – «Fatal Labyrinth» (рис. 1.6). За рік вона була адаптована до різних консолей а також вийшла на американський ринок [1].

Рис. 1.6. Гра «Fatal Labyrinth»

З першого погляду, гра була створена у популярному на той час у Японії жанрі JRPG, проте генератор лабіринтів та підземелля, предметів та їх ефектів, механіками виживання відкликали на типових представників жанру roguelike. Гра була більш простіша, а ніж інші, адже була позбавлена основної механіки ускладнення: permadeath. Ця механіка призводила до того, що персонаж повинен був померти у кінці сесії, що було частиною ігрової механіки аналогів, проте у даному випадку, такого не траплялось. Саме тому, ця гра за жанром більше підходила до спрощеного roguelite, який був більш поблажлива для гравців.

Через збільшення фанатів та кількості конкуренції, великі виробники не мали бажання відмовлятись від жанру roguelike. Саме тому, у 2005 році, культова лінійка «Pokémon» обзавелася іграми «Pokémon Mystery Dungeon: Blue Rescue Team» та «Red Rescue Team» (рис. 1.7). Не дивлячись на посередні відгуки, гра продалась тиражом у 5 мільйонів копій. Проте не дивлячись на жанр, смерть персонажа у грі була дуже ускладнена, через що більшість

сходиться у думці, що від жанру залишилось лише генерація світу та порядок ігрового процесу [21].

Рис. 1.7. Гра «Pokémon Mystery Dungeon: Blue Rescue Team»

Наступним проривом у жанрі стала гра «Ancient Domains of Mystery» (рис 1.8). Вона, так само, як і «Larn» у свій час, прагнула об’єднати у собі особливості усіх ключових ігор жанру того часу. Гру розробив програміст Томас Біскуп у 1994 році. У грі був присутній повноцінний сюжет, завдання, які вдало вписувались у концепцію, стилістику, історію та сам жанр, механіки взаємодії над персонажем та ворогами, можливість удосконалення, різноманітних предметів та ефектів, а також глобальності наслідків дій персонажа. Усе це насичувало гру великою кількість контенту та спонукало до повторного проходження, щоб дослідити ігровий світ, обираючи інші сюжетні лінії та вибори. Усі розробники вважали «Ancient Domains of Mystery» еталоном жанру, та намагались максимально адаптувати механіки та нововведення у своїх іграх [19].

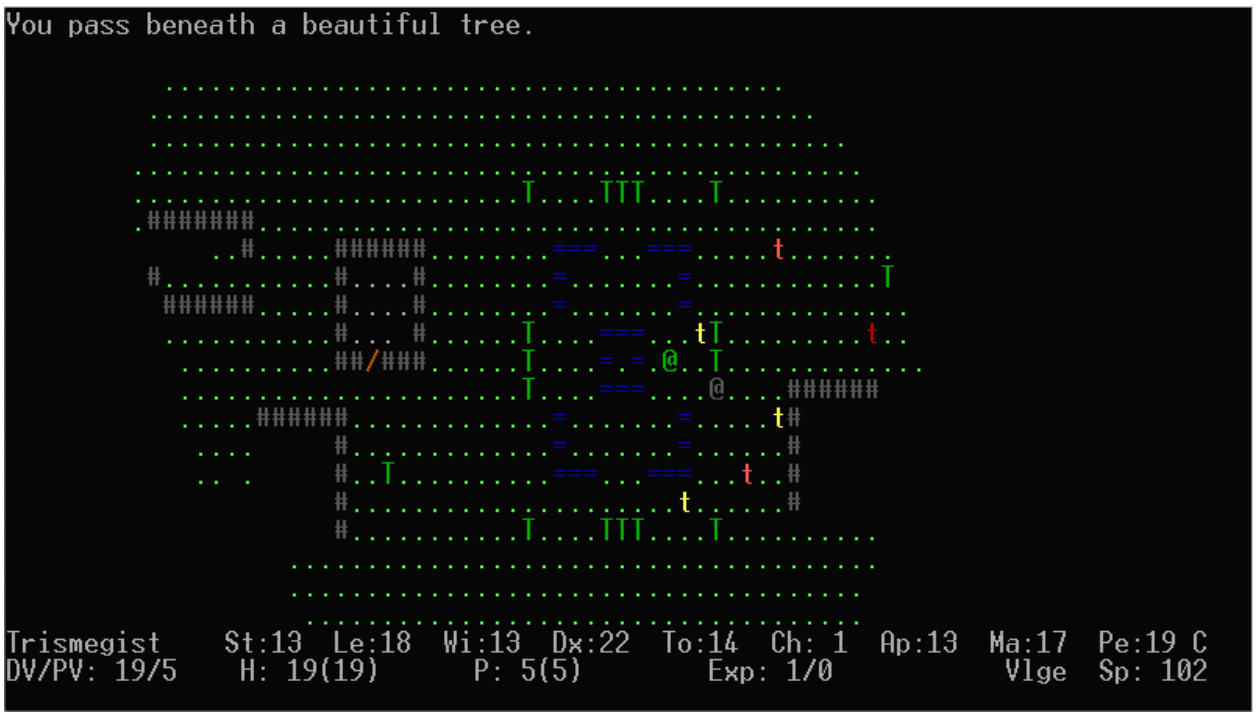

Рис. 1.8. Гра «Ancient Domains of Mystery»

На початку, гра мала примітивну символьну графіку, але згодом, у 2002 році, вийшла версія гри «ADOM Deluxe» на платформі «Steam». У грі було змінено та удосконалено багато всього. Наприклад графіка стала повноцінною, додано взаємодію гравців у багатокористувацькому режимі, розроблено багато режимів для гри, реалізовано налаштування різних ускладнень для гри, досягнення та інше (рис. 1.9).

Рис. 1.9. Гра «ADOM Deluxe»

З розвитком технологій у іграх покращувалась графіка, варіації дозвілля у грі та гумор. Саме остання характеристика зберігалась та удосконалювалась протягом усього часу. Так ось у 2011 році, команда розробників «Gaslamp Games» створила гру «Dungeons of Dredmor», яка позиціонувала себе, як провідник у жанр roguelike. Це обумовлювалось простотою в користувацькому інтерфейсі, приємною графікою, легкому гумору та відносно простому геймплеї [18].

Рис. 1.10. Гра «Dungeons of Dredmor»

Усі наступні проєкти, як і увесь час, удосконалювались та поєднувались з різноманітними жанрами. Наразі існує багато варіацій roguelike у поєднанні з різними темами та жанрами. Наприклад поєднанням з комбінаціями з покеру, як у грі «Balatro», пінг-понгом – «Peglin», вводом тексту з клавіатури для подолання лабіринтів «KKCKC» та інші безумні та нетривіальні ідеї.

Підсумовуючи, жанр зберігав свою структуру до 2020-х, згодом розробники розділились на створення класичних roguelike та експериментаторів, котрі залишили від жанру тільки високий фактор випадковості, процедурної генерації та своєрідних механік подолання складнощів.

1.2. Характерні риси Roguelike-ігор

Жанр roguelike пройшов велику історію, за яку сформував багато правил, які і встановили цей жанр, як самостійний. Давайте проаналізуємо більш детальніше кожний аспект цього жанру [13;18;22;24]:

1. процедурна генерація. Створення постійно нового рівня, поверху, підземелля. При кожному запуску гри, проходження до нового рівня або локації, гра генерує нові, неповторні локації та їх комбінації. Процедурна генерація може бути реалізована у декількох формах: створення структур випадкової форми та розмірів з коридорами або ж створення послідовно однакових кімнат за розміром кімнат з різним наповненням, що формує рівень підземелля. У будь-якому випадку, це нова комбінація структур, зазвичай має певний «seed» – номер випадкового генератора;

2. випадковість. Одна з основних характеристик жанру. Даний параметр може впливати на усю грабельну складову. Це може бути як генератор рівня, який використовує випадкове число для задання «seed» генератора, таблиця випадковості випадіння предметів та їх рідкість. Також впливає на випадкові ефекти предметів, подій оточення, варіантів атаки монстрів та інше. Впливає на рівень азартності гравця;

3. перманентна смерть. Це поняття означає, що кожна смерть під час ігрового процесу, розпочинає проходження певного забігу заново, вже в іншій генерації. Це додає цінності ігровим ресурсам та рівню життя персонажа, та підтримує напругу гравця на достатньому рівні. Деякі представники жанру мають елементи збереження певного процесу між сесіями, наприклад збереження спеціальної валюти для відкриття нового контенту, полегшення забігу, тощо;

4. покроковий геймплей. Більшість roguelike ігор використовує принцип виконання однієї дії за крок, що допомагає гравцям продумувати наперед свої дії;

5. битви. В більшості представниках жанру присутні бійки з монстрами або іншими істотами, задля того, щоб пройти далі по локації. Битви

можуть бути реалізованими будь-яким чином, але присутність ворогів та здоров’я підвищує рівень напруги та удосконалює ігровий процес. Вороги дозволяють персонажу удосконалюватись, вивчати нові їх варіацій, атаки, поведінку тощо. Зазвичай, рівень складності (здоров’я, броня, сила атаки, сила штучного інтелекту) збільшується або за часом, або за кількістю пройдених поверхів;

6. самостійне вивчення ігрового світу. Ігри спонукають до вивчення усього, що оточує гравця. Часто, у грі розробляють велику кількість унікальних механік, з якими гравець стикається випадково, що заохочує його до вивчення принципу роботи певних механік. Так само це працює з предметами, їх позитивними та негативними характеристиками, щоб гравець, методом проб та помилок зміг зрозуміти принцип дії, комбінації та ефекти кожного з них;

7. предмети. Активні чи пасивні предмети посилюють персонажа або надають певних ефектів. Знаходиться, купується або отримується у нагороду. Залучає гравця до удосконалення та підготовці до нового, більш складного рівня;

8. різноманітність проходження. Залучення гравця до пошуку аналогічних варіантів проходження. Наприклад вивчення всього поверху, або лише її частини для просування далі до цілі або можливість знаходження пропуску частин мапи чи ухиляння від певних сутичок з ворогами;

9. балансування. Важливий аспект у будь-якій грі. Цей аспект впливає на складність та можливість проходження забігу, можливостей його облегшення чи погіршення. Зазвичай, розробники на основі даних гравців, регулярно змінюють певні предмети, щоб збалансувати гру;

10. реіграбельність. Можливість повторного проходження гри. Залучення гравцям повторити забіг за допомогою: відкриття нових предметів, персонажів, кімнат, ворогів, босів, кінцівок тощо, задля залучення гравця до повторного проходження забігу або гри;

11. збирання ресурсів. Елемент менеджменту ресурсів у грі задля

купівлі спорядження, предметів, відкриття додаткових кімнат та інше;

12. сюжет. Розуміння цілі гри, мотивації проходження рівнів, розвитку персонажа та кінцівки, але не завжди потрібен;

13. завдання та досягнення. Елемент заохочення гравця проходити гру нестандартними способами, виконувати певні випробування для отримання нагороди.

Отже усі ці компоненти, так чи інакше, впливають на приналежність до ідеальної варіації roguelike гри. Реалізовуючи ці аспекти, ми збільшуємо основний параметр для ігор – реіграбельність, що досягається шляхом заохочення вивчення нового.

1.3. Аналіз ігор-аналогів

Для створення власного проєкту, необхідно ознайомитись з аналогічними іграми обраного нами жанру зі схожими механіками, які ми використаємо в майбутньому. Аналіз будемо робити на самих успішних іграх жанру roguelike, а саме: «The binding of Isaac: Rebirth», «Enter the Gungeon», «Risk of Rain 2», «Rogue Legacy 2» та «Brotato». В усіх цих іграх обрано видання з усім доступним додатковим контентом від розробника. Це зумовлено тим, що наприклад у грі «The binding of Isaac: Rebirth», доступний додатковий контент, який у щонайменше збільшує кількість персонажів, локацій, предметів, ворогів, кінцівок більше ніж у 2 рази, та дозволяє більш об’єктивно оцінити повний, готовий продукт.

Назва: The binding of Isaac: Rebirth [16].

Розробник: Едмунд МакМіллен, Nicalis, Inc, випущена у 2011 році, підтримка проєкту ведеться й у сьогоденні.

Жанр: roguelike, action.

Мова програмування: C++.

Ігровий рушій: редактор скрипів і логіки геймплею Lua та основний рушій Nicalis Engine. Раніше використовувався Adobe Flash у парі з ActionScript.

Підтримувані платформи: Windows, MacOS, Linux, PlayStation 4/5/Vita,

Xbox One/Series X/S, Nintendo Switch/3DS, Wii, IOS.

Переваги:

1. ігровий світ. Гра має унікальну атмосферу та тематику, що дозволило розробнику створити величезну кількість секретів, пасхалок, посилань на релігійних персон та їх предметів (рис. 1.11);

2. реіргабельність. Гра налічує 34 персонажі, 640 досягнень, тисячі предметів та варіацій кімнат для генерації, альтернативні шляхи проходження рівнів, 4 режимами гри, системою викликів та секретів, підтримкою модифікацій. У грі настільки багато контенту, що для її повноцінно проходження потребується щонайменше 800 годин;

3. система предметів. Предмети у цій грі не просто існують, а постійно комбінуються одне з одним. Гра має величезні формули прорахунку статистик персонажа, що дозволяє розширювати поле з експериментами у комбінації багатьох предметів, що залучає гравця до самостійного вивчення усіх ефектів предметів. Це зумовлено тим, що без додаткових модифікацій, ми не дізнаємось, що предмет робить не підібравши і не проекспериментувавши з ним;

4. модифікації. Гра офіційно дозволяє створювати та завантажувати модифікації, що розширяє потенційний контент у грі. У певний період розробки гри, одна з глобальних модифікацій була перероблена та офіційно додана до гри, як за основний сюжет;

5. висока продуктивність. Гра запускається та працює навіть на дуже слабких системах;

6. кооперативний режим. В останніх оновленнях було додано онлайн режим, задля збільшення охоплення не самотніх гравців. Цей елемент збільшує задоволення від процесу у потенційних гравців.

Недоліки:

1. проблеми з модифікаціями. Бібліотека для створення модифікацій давно не оновлювалась та не оптимізувалась, через що, велика кількість модифікацій або їх глобальність може створювати проблеми у роботі гри;

2. нестабільний баланс. Багато предметів не мають користі, а деякі дозволяють душе швидко завершити забіг просто своєю наявністю. Гра постійно оновлюється, але проблемні предмети та механіки залишаються через оновлення;

3. можливість «зламати гру». Деякі з предметів можуть нелогічно комбінуватись, даючи гравцю можливість отримати нескінченну кількість ресурсів, що дозволяє йому пройти гру або отримати більше 200 досягнень за один забіг;

4. платний додатковий контент. Купуючи гру, користувач не отримує останню версію гри, адже вона доступна лише при купівлі усіх доповнень, що збільшує ціну повноцінної гри. Також більшість доповнень не працюють одне без одного.

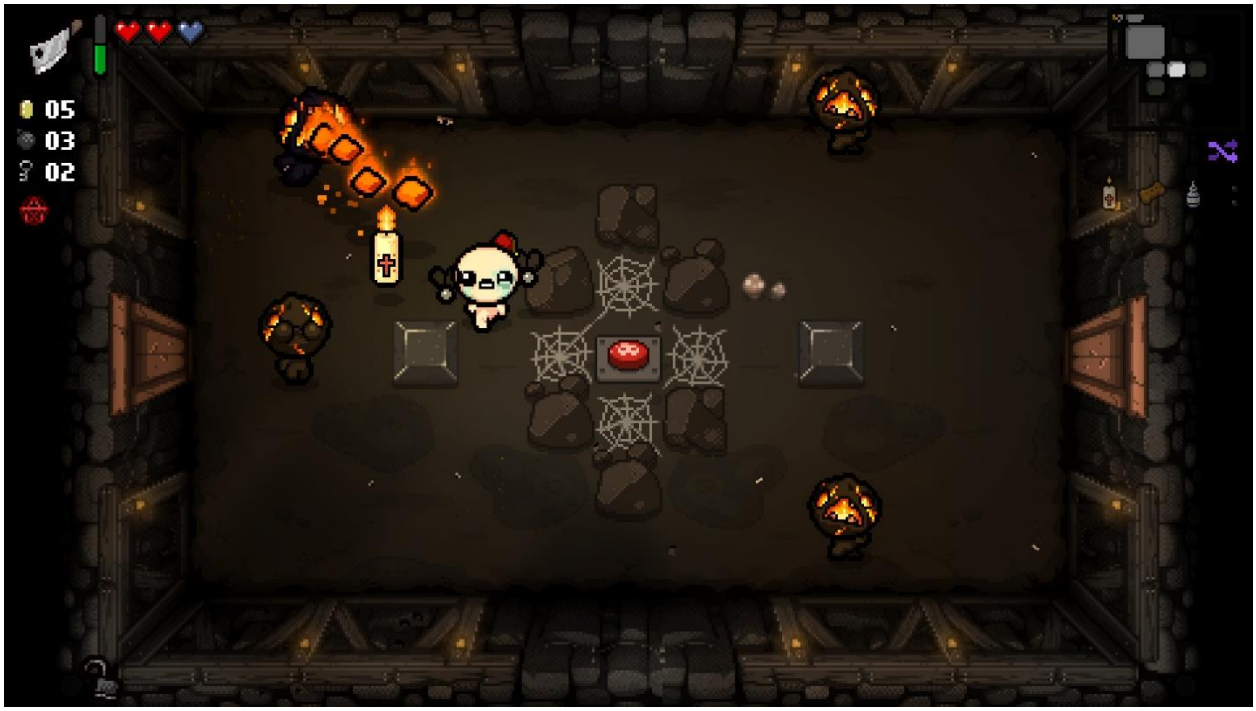

Рис. 1.11. Гра «The binding of Isaac: Rebirth»

Назва: Enter the Gungeon [4].

Розробник: студія Dodge Roll, випущена у 2016 році, підтримка проєкту ведеться й у сьогоденні.

Жанр: roguelike, action, adventure, indie.

Мова програмування: C#.

Ігровий рушій: Unity.

Підтримувані платформи: Windows, Linux, MacOS, PlayStation 4/5, Xbox One/ Series X/S, Nintendo Switch.

Переваги:

1. ігровий світ. За ідеєю розробників, гравець потрапляє у світ зброї, у якому усе персонажі та зброя комічно зображують абсурдну реальність живої зброї. Наприклад на перших рівнях ми боремося з великими патронами та гільзами, які використовують зброю (рис. 1.12);

2. арсенал. Гра налічує велику кількість активних та пасивних предметів, які впливають на дозвіл гравця. Більшість зброї може комбінуватись отримуючи сильні або неочікувані, комічні ефекти. У кожної зброї є опис, який може бути пов’язаний з реальним світом, але гумористично стилізований під факти або особливості предмету;

3. безкоштовний додатковий контент. Задля отримання усього контенту у грі, користувачу не треба витрачати зайві кошти, потрібно лише мати копію основної гри;

4. динамічний геймплей. Гра має елементи «bullet hell», за допомогою яких, на екрані дуже часто відображаються десятки, а то й сотні снарядів, що пришвидшує темп гри, завіює велику кількість реакції, та заохочує до покращення навичок володіння персонажем;

5. оптимізація. Не зважаючи на величезну кількість активних елементів, снарядів та ворогів, гра працює навіть на слабких пристроях;

6. можливість альтернативного проходження. Гра має варіанти різних кінцівок та можливостей доходження до фінальних цілей.

Недоліки:

1. не існує мобільної версії. Гра доступне лише на настільних платформах, що зменшує потенційну аудиторію гри;

2. складність. Гра вимагає високого рівня концентрації та досвіду у подібних проєктах, має достатньо високий рівень входження. В середньому гра проходиться за 300 годин, тому що має невелику кількість персонажів та кінцівок;

3. відсутність онлайн режиму. Гра з другом доступна лише на локальному рівні, що унеможливлює отримати задоволення, якщо користувачу захочеться проходити гру не самостійно;

4. відсутність повноцінної підтримки модифікацій. Для того, щоб використовувати користувацькі доповнення потрібно використовувати сторонні програми або клієнти гри.

Рис. 1.12. Гра «Enter the gungeon»

Назва: Risk of Rain 2 [11].

Розробник: студія Hopoo Games, випущена у ранній доступ з 2019, у повноцінний з середини 2020 року.

Жанр: roguelike, action, indie.

Мова програмування: C#.

Ігровий рушій: Unity.

Підтримувані платформи: Windows, Linux, PlayStation 4/5, Xbox One/ Series X/S, Nintendo Switch.

Переваги:

1. 3D світ. На відміну від класичних представників жанру, гра реалізована у великому, процедурно згенерованому 3D просторі, що відрізняє її на фоні інших проєктів (рис. 1.13);

2. динамічний гейплей. Постійне збільшення складності та простору, звідки на гравця може напасти ворог, тримає атмосферу у напрузі та вимагає від гравця реакції та необхідності діяти швидко;

3. предмети. У грі присутня величезна кількість предметів для зброї, яка гармонічно поєднується та комбінується;

4. кооперативний режим. Можливість проходити гру у сесії до 4 осіб. Порівнюючи з аналогами має найбільш якісно реалізований онлайн режим;

5. активна підтримка модифікацій. Спільнота постійно створює нові модифікації, та удосконалює сторонній додаток для цього.

6. персонажі. Гра налічує 14 персонажів, кожен з яких має особливі ігрові здібності та стиль.

Недоліки:

1. немає офіційної підтримки модифікацій. Незважаючи на величезну кількість контенту, створеною спільнотою, для її реалізації використовується додаткове програмне забезпечення;

2. складність на початку. У грі слабо реалізовано навчання, через що потенційно новим гравцям складно буде зрозуміти механіки та принцип гри;

3. технічні проблеми. Так як проєкт достатньо свіжий, розробники ще не встигли повноцінно оптимізувати свій продукт. Найбільше всього вразливі онлайн сесії, які часто відвисають та викликають відключення;

4. відсутність мапи. Великі випадково згенеровані локації, у яких можна легко загубитись через відсутність мапи.

Рис. 1.13. Гра «Risk of Rain 2»

Назва: Rogue Legacy 2 [14].

Розробник: студія Cellar Door Games, випущена у 2022 році.

Жанр: roguelike, action, adventure, indie, RPG.

Мова програмування: C#.

Ігровий рушій: Unity.

Підтримувані платформи: Windows, PlayStation 4/5, Xbox One/Series X/S, Nintendo Switch, SteamDeck.

Переваги:

1. концепція. Гра проходить у стилі 2.5D платформеру середньовіччя, в якому гравець після кожної смерті обирає собі нового головного героя, який унаслідує унікальні риси минулого персонажа (рис 1.14).

2. різноманітність ігрових класів. У грі представлено 15 унікальних ігрових класів, які заставляють гравця змінювати свій стиль гри під кожен з них;

3. прогрес. У грі присутні елементи збереження прогресу між забігами. Гравець можете удосконалювати глобальні навички гравця, а також удосконалювати кожен окремий клас, граючи за нього;

4. естетика. Гра має плавні анімації та приємний дизайн, що покращує процес поглиблення у гру;

5. геймплей. Гра поєднує у собі елементи бою разом з платформінгом та стратегічним мисленням.

Недоліки:

1. висока складність. Новачкам буде складно прогресувати у грі, так як зброя, зовнішній світ та вороги вимагають достатнього рівня концентрації та координації персонажа;

2. генерація. Багато елементів структур можуть повторюватись через невелику базу для генерації;

3. відсутність кооперативного режиму та модифікацій. Гра націлена лише на проходження однією особою. Також розробники не дають інструменти для створення модифікацій у своїй грі;

4. баланс. Дуже багато проблем створюють класи, так як деякі з них занадто сильні, що змушує гравців завжди обирати одні і ті ж класи для спрощення процесу гри.

Рис. 1.14. Гра «Rogue Legacy 2»

Назва: Brotato [9].

Розробник: Blobfish, випущена у 2023 році.

Жанр: roguelite, action, arena shooter.

Мова програмування: C++, GDScript.

Ігровий рушій: Godot.

Підтримувані платформи: Windows, Android, IOS, PlayStation 4/5, Xbox One/Series X/S,

Переваги:

1. геймплей. Швидке та динамічне проходження хвиль на арені, ідеально підходить для коротких сесій (рис. 1.15). Має декілька режимів гри та можливість нескінченного проходження;

2. дизайн. Простий дизайн дозволяє запускати гру на мобільних пристроях та слабких комп’ютерах;

3. кількість персонажів. У гри присутні 62 персонажі з унікальними характеристиками та стилями гри.

4. реіграбельність. Через величезну кількість персонажів та предметів, у гравця є масивна зона для комбінування та створення власних збірок предметів. Також у грі присутні системи викликів, що урізноманітнюють ігровий процес;

5. інтуїтивність. Усі дії у грі прості та інтуїтивні, зброя стріляє самостійно, треба лише збирати предмети та ухилятись від ворогів та їх снарядів.

Недоліки:

1. відсутність сюжету. Гра фокусується лише на її механічній частині, нехтуючи сюжетом та історією персонажа та мотивації його дій;

2. монотонність. Гра дуже однотипна, вороги та їх генерація передбачувана, карти відрізняються лише фоном;

3. відсутність кооперативу та модифікацій. Розробник ще не реалізував механіку кооперативної гри та додавання користувацьких модифікацій;

4. складність балансування. Через величезну кількість характеристик, персонажів та предметів, складно все тримати на однаковому

рівні.

Рис. 1.15. Гра «Brotato»

Отже на основі цього ми зрозуміли, що усі вище перераховані ігри мають такі риси, як: наявність сесій та перманентної смерті персонажа, наявність великої кількості випадкових подій, акцент на удосконаленні власних навичок, вивченні ворогів для полегшення ігрового процесу, у більшості процедурна генерація рівнів, велика кількість предметів, їх комбінацій та ігрових персонажів з унікальними стилями та механіками.

Висновки до розділу 1

Отже, у першому розділі ми проаналізували поняття жанру roguelike, розглянули історію створення та розвитку жанру, основних ігор представників та їх нововведення, виділили основні характерні риси жанру та проаналізували сучасні ігри-аналоги.

Roguelike – це жанр ігор, характерний покроковим ігровим процесом, процедурною генерацією рівнів, боєм з монстрами та іншими істотами, перманентною смертю та реіграбельністтю.

Ігри цього жанру можуть бути різноманітними, головне, щоб були присутні більша кількість цих характеристик.

Roguelite – піджанр roguelike, характерний спрощенням принципу перманентної смерті та зменшенню складності гри.

Ігри саме такого жанру вперше з’явились у Японії, при спробі розвинути жанр roguelike у країні.

В основному, ігрові студії та розробники намагаються зберегти автентичність жанру, але додаючи до нього унікальні механіки та методи гри.

РОЗДІЛ 2
МОЖЛИВОСТІ ROGUELIKE-РОЗРОБКИ В GODOT ENGINE

2.1. Огляд рушія Godot Engine

Godot Engine – це сучасний, багатоплатформний, потужний ігровий рушій з наявністю відкритого коду, орієнтований для створення 2D та 3D проєктів [21]. Програмування елементів здійснюється за допомогою таких мов програмування як: GDScript, C++, C#, VisualScript, а також працює на основі таких платформ, як: Windows, Linux, maxOS, iOS, Android [1].

Розробники рушія, а саме Джуан Лінієтська та Аріель Манзур, при розробці закладали ідеї повної свободи для користувачів, під час створення власних проєктів. Це проявляється через відсутність нав'язливості обмежень чи шаблонів, даючи користувачу повний контроль над процесом та простір для творчості [17].

Основою побудови у проєктах є вузли. За допомогою них встановлюються зв'язки між компонентами, програмовим кодом та іншими вузлами. Кожен з вузлів, може зберігати дані різних типів, наприклад: зображення, анімації, скрипти, моделі, тіла, об'єкти, фізику з взаємодією та інше [5].

Найбільш функціональним є вузол «Node2D», що в свою чергу потрібен для створення 2D чи 3D сцен. Він застосовується до усіх об'єктів, що хоч якось можуть рухатись, мати колізію, анімацію масштаб чи обертання. Окремо, цей вузол не має візуального відображення, проте задає параметри базової системи координат, у якій працюють дочірні, допоміжні вузли. Таким чином, майже будь-який елемент сцени, наприклад персонаж, ефект чи елементи простору, можуть розміщуватись та трансформуватись відносно головного вузла [1, 23].

Вузол «Node2D» використовує для цього певні властивості [5, 6]:

1. position. Визначає координату вузла для об'єкта у певному, локальному просторі. Має аналогічну глобальну версію: «global_position»;
2. rotation. Визначає обертання вузла у одиницях вимірювання радіан. Має аналогічну глобальну версію: «global_rotation»;

3. scale. Визначає масштабування вузла об’єкта відносно осей X та Y. Має аналогічну глобальну версію: «global_scale»;

4. z_index. Визначає порядок, у якому будуть вимальовуватись об’єкти на сцені (визначає за допомогою системи шарів);

5. pivot_offset. Визначає позицію для зміщення центру обертання чи масштабування об’єкта;

6. transform. Визначає повну матрицю змін та трансформації об’єкта, включаючи позицію, обертання та масштаб, проте потребує окремого об’єкта типу: «Transform2D».

Завдяки своїй універсальності, «Node2D» використовується як контейнер для створення більш спеціалізованих вузлів, такий як [5, 6]:

1. Sprite2D. Малювання спрайтів;

2. CollisionShape2D. Відтворення колізії на декількох рівнях з налаштуванням шарів взаємодії;

3. AnimatedSprite2D. Створення анімації на основі тайлів, взаємодій тощо;

4. Area2D. Створення зон для взаємодій об’єктів та анімації.

Для створення користувацького інтерфейсу, головним вузлом є – «Control». Він обробляє позицію та розміри об’єктів на екрані, введення даних периферії, підтримує розмітку та розташування контейнерів на екрані, може використовувати задані користувачем теми та має обширний список властивостей, для більш детального налаштування реакцій на інші елементи проєкту [2].

Для поєднання та взаємодії вузлів використовуються сцени, які зберігають у собі структуру вузлів, дозволяючи їх взаємодіяти один з одним у рамці виконуваної сцени. Вона зберігається окремим файлом та може бути використана у інших сценах (методом виклику чи шаблону) чи як самостійний об’єкт [21]. Наприклад у нас є сцена з гравцем, котру ми можемо використовувати як шаблон, щоб відобразити гравця на сцені конкретного рівня.

Для реалізації функцій логіки взаємодії використовуються скрипти. Вони можуть бути призначені до вузлів, створюючи їм певну поведінку, що дозволяє реагувати на події, виклики, керувати анімацією, обробкою введення тощо. Скрипти успадковують класи вузлів, до яких вони прив'язані а також можуть звертатись до функцій інших вузлів, створюючи взаємодію між об'єктами та дозволяє користувачеві організувати структурне дерево файлів [6].

Інтерфейс програми виглядає наступним чином [1, 5]:

1. панель сцен. Розташований у лівій верхній частині екрану, дозволяє користувачу додавати, редагувати та пересувати вузли. Програма одразу відображає дерево зв'язку вузлів, підключених до них скриптів та анімацій. За наявності відображаються символами (рис 2.1);

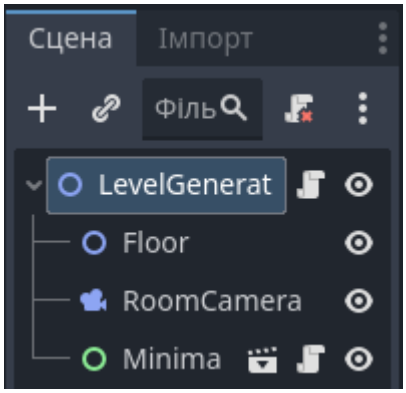

Рис. 2.1. Редактор дерева сцени

2. інспектор. По стандарту розташований у правій частині екрану. Тут відображаються усі доступні характеристики вузла. Також ми можемо редагувати вузол на різних рівнях, які відсортовані по вкладках, а також додавати скрипти. Із основного: характеристики скрипту, типу вузла, «CanvasItem» для візуальної зміни та вкладка змін опису та активного скрипту вузла. Якщо додати змінні у скрипти через метод «@export» (див. 2.2), вона з'явиться тут і буде готова до редагування змінної або додавання потрібного елемента (рис. 2.2). Також ви можете у вкладці вузол додати потрібний сигнал із списку запропонованих, задля реалізацій логіки, наприклад вбудовані системні функції та функції «CanvasItem» та інше (див. 2.2).

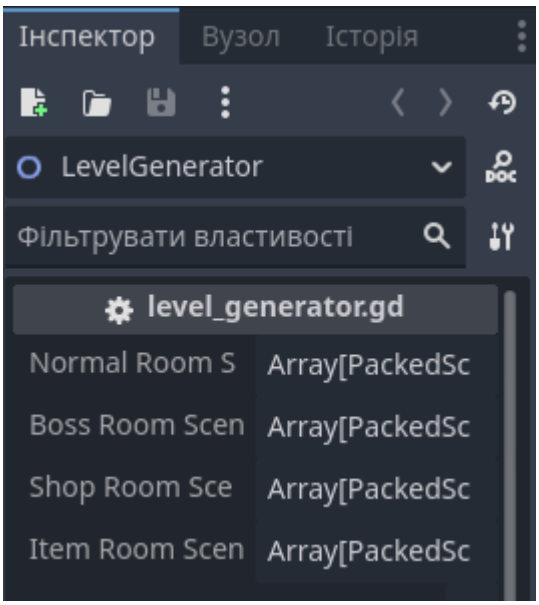

Рис. 2.2. Інспектор

3. файлова система. Розташовано у лівій нижній частині. Демонструє структуру файлової системи проєкту. Дозволяє користувачу у режимі «Drag&Drop» завантажувати файли та шляхи до інспектора, панелі сцен, активного коду, тощо. Також можна редагувати дерево, з автоматичною зміною шляхів у скриптах (рис. 2.3);

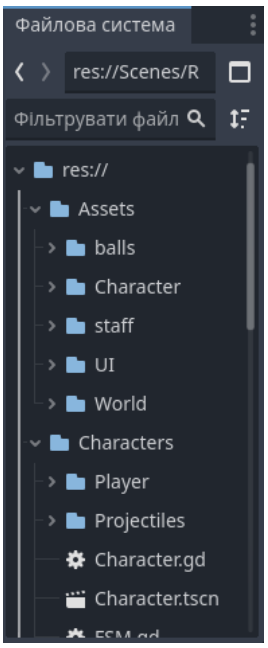

Рис. 2.3. Файлова система

4. редактор. Згори користувач обирає активну вкладку для редагування. Перші 2 вкладки «2D» та «3D» призначені для редагування та перевірки відображення явних об’єктів, їх хітбоксів та місць розташування. У

вкладці «Script» відбувається зміна програмового коду обраного скрипту. Зліва зверху список усіх файлів проекту розширення «.gd», нижче список функцій обраного скрипту і основним вікном цієї частини – текстовий редактор. Також між основною частиною та вибором вкладки розташовані активні відкриті сцени. Вкладка «Game» демонструє гру у редакторі, без окремого запису, а вкладка «AssetLib», демонструє вбудовану бібліотеку ресурсів Godot Engine, а саме: шаблони, текстури, плагіни тощо (рис. 2.4);

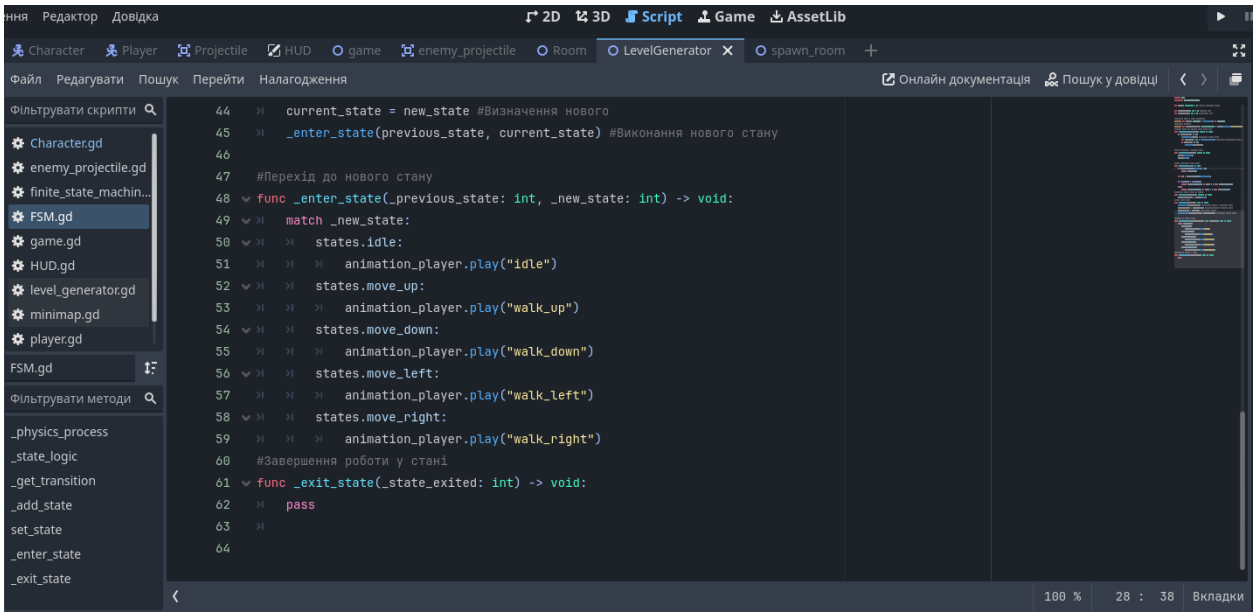

Рис. 2.4. Редактор

5. нижня панель. Вона розташована по середині знизу. У першій вкладці «Вивід» відображаються помилки, попередження, вивід повідомлень, результат виконання команди «print()». «Засіб діагностики» більш детально демонструє змінні, які використовуються у грі проєкті під час запуску та роботи програми, помилки, використані помилки тощо. Останні вкладки використовуються для редагування звуків, шейдерів, анімацій, наборів тайлів для обраного вузла (рис. 2.5);

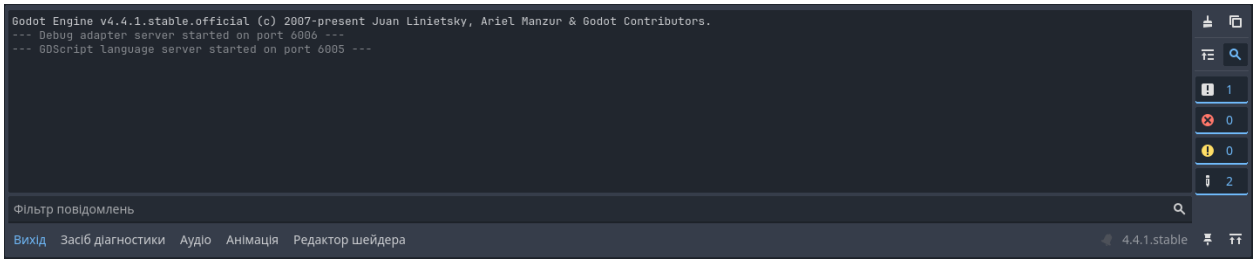

Рис. 2.5. Нижня панель.

6. кнопки відтворення. Розташована у правій верхній частині. Використовуються для збирання та запуску загально проєкта або обраної сцени для перегляду роботи обраних фрагментів (рис. 2.6).

Рис. 2.6. Кнопки відтворення

2.2. Скриптова мова GDScript

GDScript це високорівнева, динамічно типізована, скриптова мова програмування, спеціально створена для рушія Godot Engine. Завдяки повній інтеграції у рушій, вона має прямий доступ до вузлів, сцен, ресурсів тощо, а також, повноцінно функціонує на усіх підтримуваних рушієм платформами [17].

Розберемо основні приклади синтаксису скриптової мови програмування GDScript [21, 1, 5, 6]:

1. змінні та типізація. Оголошення відбувається за допомогою команди «var», після якої оголошується назва змінної, далі через знак «:» тип даних, або самі дані через комбінацію знаків «:=». У першому прикладі оголошується змінна позиції камери як тип даних, у другому як певне початкове значення (рис. 2.7).

```
var camera_target_position: Vector2
var camera_speed := 5.0
```

Рис. 2.7. Приклад оголошення змінних

Також перед оголошенням модна використати конструкції «@export» та «@onready». Перша з них дозволяє показувати змінну одразу у інспекторі редактора для того, щоб ми могли її зручно та швидко змінювати, без потреби редагування з коду. Також це зручно для оголошення додаткових, окремих сцен. Друга конструкція допомагає визначити змінну одразу, як вузол увійде до сцени, тобто після конструкції «ready()», що виконує первинний код скрипту (рис. 2.8).

```
@onready var camera: Camera2D = $RoomCamera
@export var minimap: Control
```

Рис. 2.8. Приклад використання додаткових конструкцій

Разом з цим, виконується типізація змінної. Якщо ми одразу введемо значення, тип пуде динамічним, і може змінюватись при зміні значень. Але якщо ми одразу зазначили тип змінної, він стане статичним, і буде видавати помилки при спробі зміни типу даних через значення.

2. функції. Створення функцій виконується за допомогою використання конструкції «func», після якої ми оголошуємо назву функції, значення які вона приймає (у дужках), та саме тіло функції. Виокремлення виконано за допомогою табуляції. Для того, щоб звернутись до функції, потрібно написати її назву та, за потреби, вказати змінні для подачі. Разом з цим, системні та користувацькі функції відрізняються за допомогою символу «_» на початку, який присутній у системних (рис. 2.9). Створення нових функцій з цим позначенням не призведе до автоматичного виконання функції за потрібними правилами;

```
func _ready():
    call_deferred("_init_minimap")
```

Рис 2.9. Приклад функції та виклику функцій

3. класи. На початку коду автоматично створюється оголошення класу за допомогою команди «extends», що визначає призначення скрипта до певного вузла (рис. 2.10);

```
extends Node2D
```

Рис. 2.10. Оголошення класу

4. взаємодія з вузлами. Скриптова мова програмування дозволяє звертатись до об’єктів, елементів та функцій інших вузлів. Для цього нам треба зазначити шлях об’єкту, до якого ми звертаємось та потрібку дію (рис. 2.11).

```
minimap.draw_rooms(used_positions)
```

Рис. 2.11. Звернення до функції іншого вузла

Також ми можемо напряму змінювати характеристики вузлів, якщо

звернемось до їх характеристики у рамках однієї сцени (рис. 2.12);

```
coins_label.text = "%d" % coins
```

Рис. 2.12. Приклад звернення до характеристики вузла

5. умовні операції. Структурно, умовні операції реалізовано аналогічно до мови програмування Python. Тобто, у нас є команда, наприклад «if», після цього символ порівняння та значення. Особливим та зручним є використання конструкції «match» для великої кількості порівняння, тому що виконується швидше та займає менше місця (рис. 2.13).

```
func update_health(current_hp: int, max_hp: int):
    for i in range(max_hp):
        var heart_value = clamp(current_hp - i * 2, 0
        match heart_value:
            2:
                hearts[i].texture = full_heart
            1:
                hearts[i].texture = half_heart
            _:
                hearts[i].texture = empty_heart
```

Рис. 2.13. Приклад використання умовного оператора «match»

Також використовуються вбудовані методи. По суті, це системні функції, що починаються з символу «_». Розберемо детальніше логіку їх роботи [21, 1, 5, 6]:

- _ready(). Виконується, коли прив’язаний вузол та усі його нащадки додані до сцени;
- _process(delta). Виконується кожен кадр, але без фізики, тільки для логіки;
- _physics_process(delta). Виконується кожен фізичний крок, в залежності від створеної фізики;
- _input(event). Виконується, коли приходить введення з периферії;
- _unhandled_input(event). Подібне до «_input», але викликається пізніше у циклі;

- _enter_tree(). Виконується, коли вузол додано до сцени;
- _exit_tree(). Виконується, коли видалено зі сцени;

2.3. Переваги та недоліки використання рушія Godot Engine

Рушій Godot Engine все ще підтримується розробниками та оновлюється, проте незважаючи на велику кількість переваг, є і недоліки [1, 17, 18, 21, 1, 6]:

Переваги:

1. рушій має відкритий код, що дозволяє розробникам та ентузіастам модифікувати рушій, допомагаючи насамперед розробникам розширювати функціонал рушія, а також швидше знаходити критичні проблеми;
2. простий, зручний та інтуїтивний інтерфейс з достатньою кількістю допоміжних міток;
3. проєкти, створені на основі рушія Godot Engine, підтримуються на великому ряді популярних платформ, а саме: Windows, MacOS, Linux, Android, iOS та веб-браузери;
4. розробники акцентують увагу на швидкодії, у результаті чого, ресурси ефективно використовуються навіть на мобільних платформах;
5. рушій повністю безкоштовний, не вимагає обов’язкових сплат та лояльний до комерційних проєктів, що збільшує кількість розробників цієї платформи;
6. Godot Engine має велику кількість інструментів для розробки ігор у двовимірному чи трьох вимірному просторі. Інструменти включають у себе: редактори анімацій, музики, звуків, моделей, систем освітлення, шейдерів, колізій, тощо;
7. рушій має підтримку сторонніх плагінів для збільшення функціоналу;
8. рушій має унікальну систему сцен, яка керується вузлами, що допомагає легко комбінувати та використовувати самі сцени для відображення ігрового процесу;
9. GDScript проста та лояльна скриптова мова програмування.

Синтаксично схожа на Python, зрозуміла, проста у вивченні та розумінні;

10. вбудована система сигналів допомагає швидко створити певні ділянки взаємодії об’єктів, що полегшує процес створення проєкту;

11. вбудований редактор фото-, аудіо-, відеофайлів та анімацій;

12. основна програма має невелику вагу, швидко запускається та компілює проєкти, що допомагає розробникам зі слабкими комп’ютерами відчувати комфорт на етапі створення проєкту;

13. вбудовані функції для створення повноцінної UI складовой на основі зручних «Control» вузлів.

Недоліки:

1. у порівнянні з платними комерційними рушіями, має меншу кількість користувачів, кількості великих чи малих проєктів та ресурсів ускладнює пошук інформації та аналогічних проєктів для аналізу та навчання;

2. деякі специфічні функції можуть бути не реалізовані або вимагати додаткових дій для реалізації;

3. присутні проблеми сумісності з бібліотеками, адже скриптова мова GDScript та сам рушій Godot вимагає окремої сумісності;

4. не має великої гнучкості під час візуального програмування у порівнянні з системою вузлів;

5. режим «3D» має не настільки велику кількість функціоналу у порівнянні з аналогами;

6. редактори відео та аудіо матеріалів мають невеликий функціонал, що іноді може змусити до застосування більш професійного програмного забезпечення;

7. відсутність багатьох готових рішень та матеріалів для завантаження;

8. застарілість матеріалу в інтернеті через те, що при оновленні деякі функції змінюються кардинально, або змінюють принцип роботи, через що більшість матеріалу стає неактуальною та потребує постійного звіряння з офіційною документацією;

9. офіційна документація не має занадто детального пояснення або може бути дуже сильно обмеженою;

10. використовує мережевий стек на базовому рівні, що ускладнює реалізацію сучасних методів мережного з’єднання та передачі пакетів;

11. типізація в GDScript недостатньо строга, через що помилки можуть виникати лише під час виконання проєкту.

Проте, незважаючи на недоліки, рушій залишається потужним, сучасним інструментом для створення двовимірних та трьох вимірних проєктів. Враховуючи постійне збільшення функцій, проєктів та розробників, рушій стає чудовим вибором для безкоштовного створення та реалізації власного проєкту.

2.4. Висновки до розділу 2

Отже, у другому розділі ми: детально оглянули та проаналізували рушій Godot Engine, розібрали ключові функції та інтерфейс, розібрались із принципом створення та роботи проєкту, проаналізували особливу скриптову мову програмування GDScript, розібрали синтаксис та особливі функції мови програмування а також, виокремили переваги та недоліки використання даного рушія.

На основі цієї аналітики ми можемо прийти до висновку, що рушій Godot Engine чудово показує себе у створенні саме двовимірних проєктів, має велику кількість інструментарію та можливостей скриптової мови. Структура розробки через вузли, проєкти та скрипти легко та інтуїтивно допомагає під час створення та реалізації гри.

РОЗДІЛ 3
ПРОЦЕС РОЗРОБКИ ГРИ

3.1. Планування проєкту

Перед початком безпосереднього створення проєкту, необхідно розробити детальний план, який буде описувати реалізацію процесу створення гри:

1. ідея. Для початку, потрібно вирішити концепцію та ідею проєкту. Ми будемо створювати двовимірний проєкт, жанру roguelike, текстури у майбутній грі будуть розміру «16x16», призначений для запуску з настільних комп'ютерів. Принцип роботи гри: кожен забіг генерує певну кількість поверхів зі своїми лабіринтами, які генеруються процедурно. На кожному рівні присутні: початкова кімната (у якій гравець з'являється на початку кожного рівня, яка є повністю безпечною, тобто не матиме ворогів), звичайні кімнати, які генеруються за шаблоном, можуть містити ворогів, предмети та розхідники, спеціальні кімнати, а саме: крамниця, для купівлі предметів за ігрову валюту та скарбниця для отримання предмету в обмін на ключ при вході, а також кімнатою боса, після проходження якого відкривається шлях до неступного рівня. Чим далі заходить гравець, тим більше звичайних кімнат генерується на рівні, а також змінюється склад ворогів та босів для ускладнення процесу гри при її продовженні та пробудженні у гравця мотивації удосконалювати персонажа. Головний персонаж – чаклун, який вміє стріляти у 4 сторони. Розхідники, які з'являються у грі: серця, для відновлення здоров'я, ключі для відкриття сундуків та спеціальних кімнат та монети для купівлі предметів. Гра закінчується або при смерті персонажу (поразка, зниження показника здоров'я до 0) або якщо буде пройдено 6 рівнів (перемога над усіма босами гри);

2. організація робочого середовища проєкту. Наступним кроком створюємо порожній проєкт. Це можна зробити, натиснувши кнопку «Створити» вгорі. Вводимо назву, більше нічого налаштовувати не будемо, так як не плануємо мобільну версію застосунку (рис. 3.1).

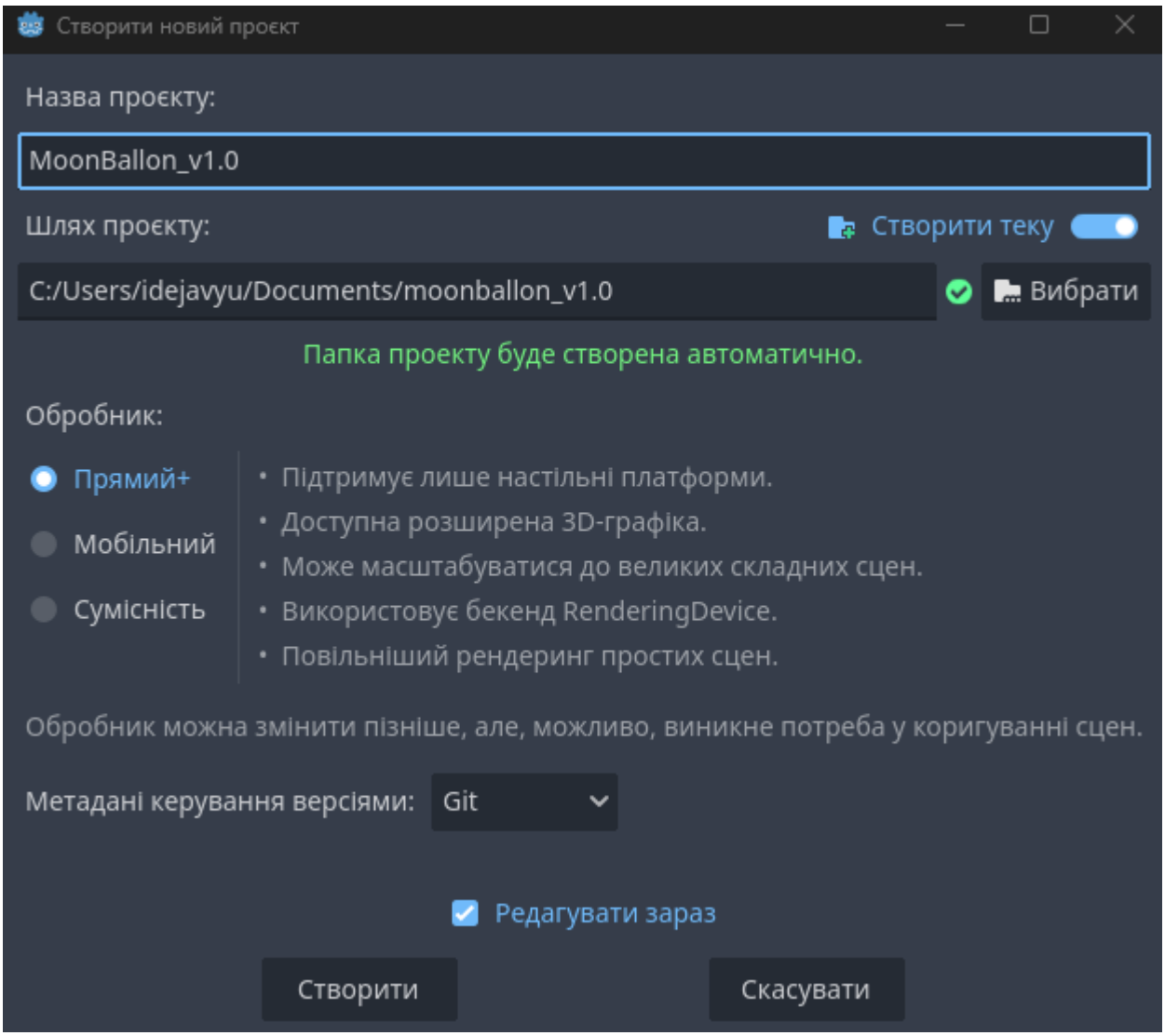

Рис. 3.1. Вікно створення проєкту

Далі нам потрібно зробити декілька важливих налаштувань. У вкладці «Налаштування проєкту» у групі налаштувань «Показ – Відео», спочатку змінюємо розміри вікна. Так як текстури у нашій грі розмірності «16x16», а орієнтовна розмірність рівнів «19x15» блоків текстур, визначаємо розмір вікна, як «304x240». Також треба змінити режим розгортання на «viewport», для того, щоб створюване грою вікно, за допомогою збереження пропорцій, могло збільшуватись та розширюватись на увесь екран (рис. 3.2).

Не виходячи з налаштувань одразу реалізуємо логіку шарів фізики. Тобто визначимо що з чим може взаємодіяти у нашій грі. Для цього переходимо до вкладки «Назви шарів – 2D фізика». На екрані з’явиться вікно, у якому відображається купа порожніх шарів (рис. 3.3).

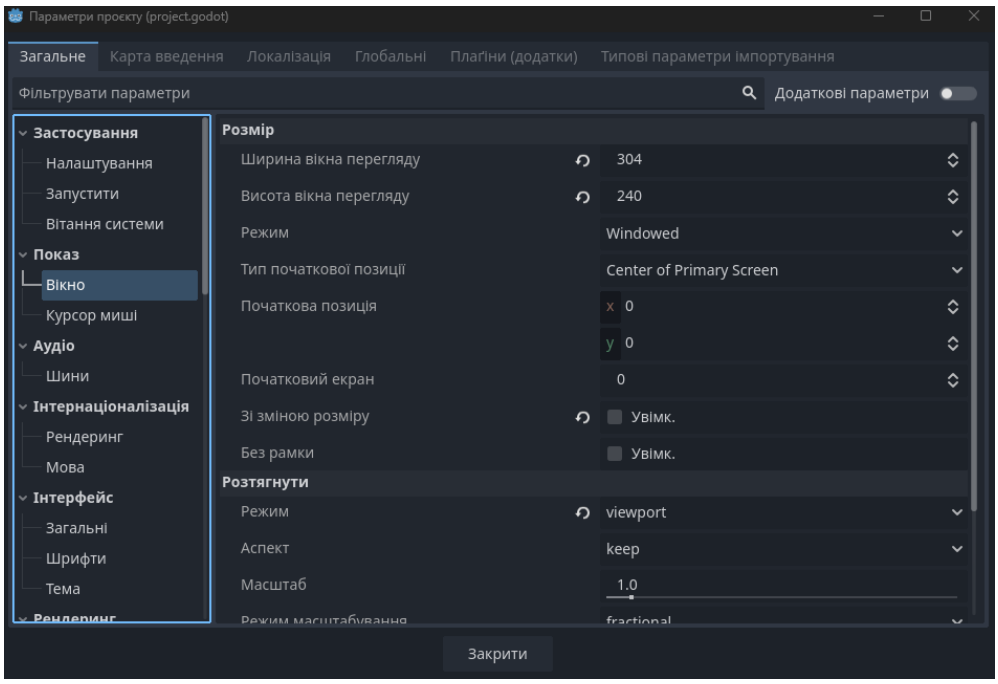

Рис. 3.2. Налаштування вікна

Для реалізації нашої задумки знадобляться лише 3 шари: «World», для створення колізії об’єктів світу, «Player» – колізія основного персонажу та його снарядів та «Enemy» аналогічно до персонажа, проте колізія стосуватиметься ворогів. Загалом у рушії взаємодія об’єктів відбувається за допомогою налаштування зав’язків між цими шарами. Наприклад для реалізації попадання снаряду у стіну, нам треба побудувати зв’язок усіх елементів «Player» зі шаром «World», що визначає увесь простір рівнів.

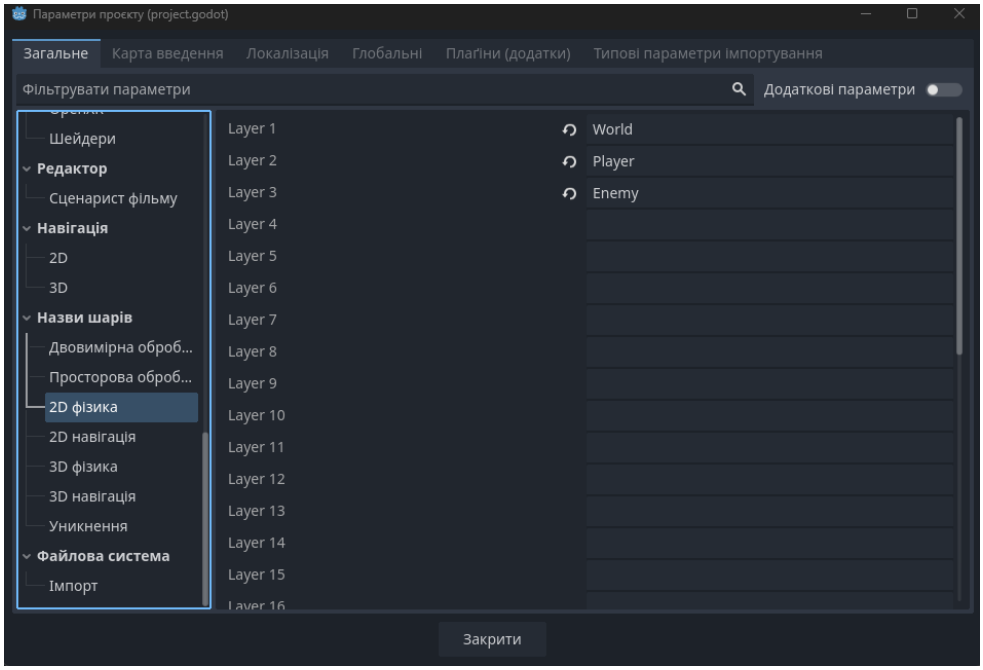

Рис. 3.3. Налаштування 2D фізики

Останнім пунктом у налаштуванні, перейдемо до карти введення. Тут ми задаємо програмі вхідні значення, на які їй треба буде реагувати. Для цього додаємо події реакцій натиску на клавіші «WASD» для руху, та стрілочок для стрільби. Параметр «Мертва зона» змінюємо до 0.1 для того, щоб натискання були майже миттєві та програмний код встигав обробляти події (рис. 3.4).

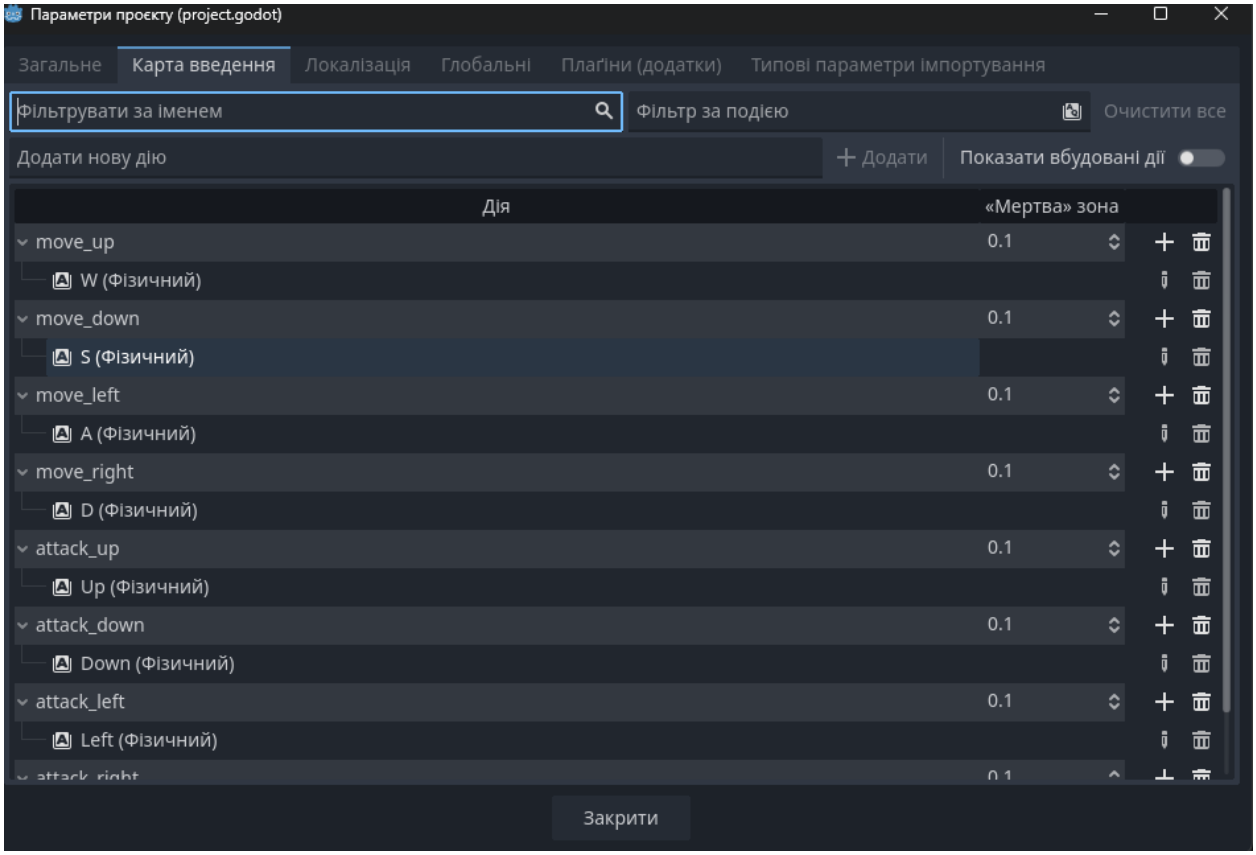

Рис. 3.4. Налаштування карти введення

Після виконання усіх цих кроків ми маємо готові налаштування нашого проєкту, що допоможе створити та реалізувати основні функції гри;

3. організація файлової системи. Для зручності та інтуїтивного розміщення файлів, створимо структуру папок. Загалом, ми будемо мати 4 папки (рис. 3.5):

a. assets. Збереження усіх зображень та «tilemap», тобто наборів текстур у вигляді сітки. Для більшої деталізації розмістимо папки, які більш точно описують зміст файлів: «Balls» – зображення усіх снарядів, «Character» – усі персонажі гри, «Staff» – зображення посохів, які використовує гравець, «UI» – іконки для створення інтерфейсу користувача, «World» – текстури для створення ігрового світу. Дана структура допомагає швидко та зручно

знаходити та звертатись до необхідних текстур;

b. characters. Збереження ключових сцен та механізмів роботи та взаємодії персонажа. У цій папці ми виокремлюємо гравця, ворогів та снаряди. Це виокремлення пов’язано з тим, що при створенні гри ми часто будемо звертатись саме до об’єктів персонажів, тому розмістили їх окремо;

c. scenes. Збереження усіх сцен, які стосуються генератора рівнів, кімнат, користувацького інтерфейсу;

d. scripts. Збереження загальних скриптів роботи гри, які не стосуються окремих сцен.

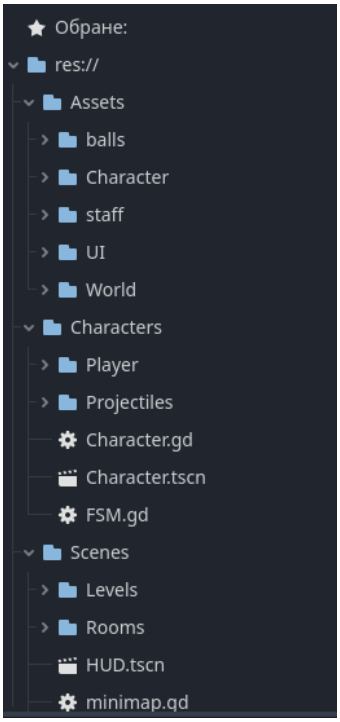

Рис. 3.5. Файлова система

За допомогою такого варіанту структури проєкту, зручно використовувати окремі сцени, текстури та скрипти за допомогою режиму «Drag&Drop»;

4. додавання ресурсів. Останнім підготовчим етапом є створення чи пошук потрібних текстур для реалізації рівнів, персонажів, снарядів, об’єків, ворогів, користувацького інтерфейсу тощо;

5. створення візуальної частини. Маючи потрібний мінімум, можемо переходити до створення анімацій персонажів, предметів, реацій, ігрового світу, макетів рівнів, користувацького інтерфейсу, тощо (див. 3.2);

6. створення ігрової логіки. Даний пункт найбільш обширний, тому що вимагає створення програмного коду взаємодій усіх об’єктів на екрані. Перед початком варто зазначити, що увесь код перевіряється на окремій тестовій сцені, для того зоб одразу ж оцінити результат та виявити критичні помилки. Розглянемо більш детальніше кроки створення ігрової логіки проєкту (див. 3.3):

a. створення логіки головного персонажу. Включає у себе пересування, можливість стріляти, колізії, виконувати анімації у потрібні моменти, мати характеристики здоров’я, шкоди, кількості розхідників, тощо.;

b. реалізація тестового рівня. Створення однієї кімнати, для реалізації колізії, взаємодії снарядів, персонажа з нею, перевірки усіх наступних елементів ігрової механіки;

c. створення користувацького інтерфейсу. Включає у себе відображення здоров'я, розхідників, місця під мапу, та інші важливі для гравця позначення;

d. реалізації процедурної генерації світу. На основі створеної базової колізії для об’єктів, реалізовуємо шаблони для генерації кімнат. Реалізуємо систему створення дверей, визначення сусідів та типів кімнат, для закриття їх під ключ. Після цього створюємо механізм, який буде генерувати рівні, на основі визначених правил;

e. додавання мапи. Реалізації мапи у користувацькому інтерфейсі для відображення кімнат та їх типів користувачу;

f. додавання розхідників. Реалізація взаємодії розхідників з персонажами, логіки їх підбирання та лімітів;

g. реалізація предметів. Включає у себе створення рідкості, правил генерації та зміни параметрів персонажа при їх підбиранні;

h. додавання ворогів та босів. Реалізація додавання у макети ворогів, логіки закриття ворогів, взаємодії їх з персонажем, нагороди за зачищення кімнати, відкриття проходу на наступний рівень.

7. оптимізація. Процес тестування гри, оптимізація окремих її

компонентів, використання ресурсів, механік тощо. Знаходження помилок та проблемних частин та методи їх вирішення (див. 3.4).

3.2. Розробка графіки та інтерфейсу

Після завантаження необхідних компонентів текстур, створимо тестовий шаблон сцени з кімнатою. Для цього у місці, на верхній панелі обираємо вкладку «Сцена». У контекстному меню буде опція «Нова сцена». Також її можна створити використовуючи комбінацію клавіш «Ctrl + N». Після цього обираємо 2D сцену. В дереві вузлів створюємо вузол типу «TileMap», який буде відноситись до основи сцени, головного вузла типу «Node2D». За допомогою нього ми зможемо графічно відобразити рівень (рис. 3.6).

Рис. 3.6. Дерево вузлів

Обираємо створений нами вузол і образу ж у головному редакторі відобразиться сітка для редагування текстур. В інспекторі у вкладці «TileMap», обираємо параметр «TileSet», та створюємо новий. Цей параметр дозволить завантажити зображення з елементами дизайну нашої гри, для подальшого його відображення. Трохи нижче зазначаємо програмі розмірність текстур гри. У нашому випадку це «16x16» (рис. 3.7).

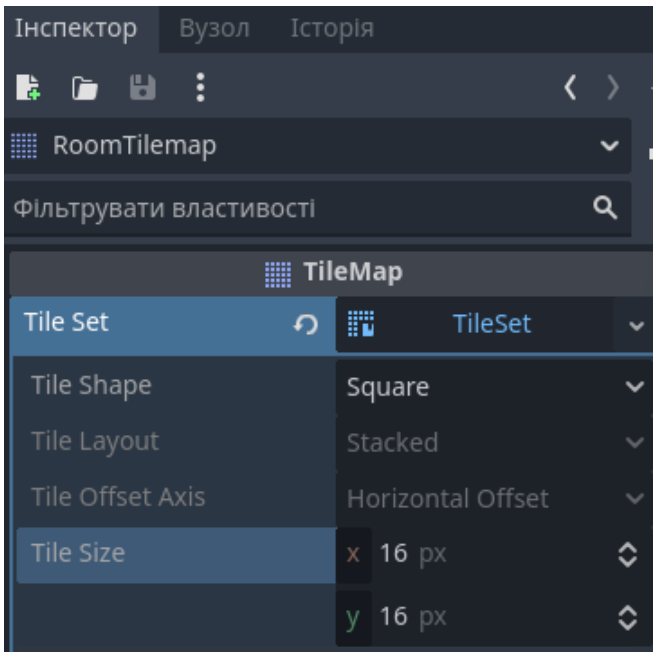

Рис. 3.7. Інспектор

Після налаштування, у нижній частині екрану з’явиться вкладка «TileSet» та «TileMap». Спочатку відкриваємо першу з них, натискаємо значок «+» та додаємо зображення з нашими текстурами. Перевіряємо розмірність області у налаштуваннях. Якщо все вказано вірно, ми побачимо сітку на нашій карті текстур. Якщо текстури використовуються з різних карт, додаємо та аналогічно налаштовуємо їх (рис. 3.8).

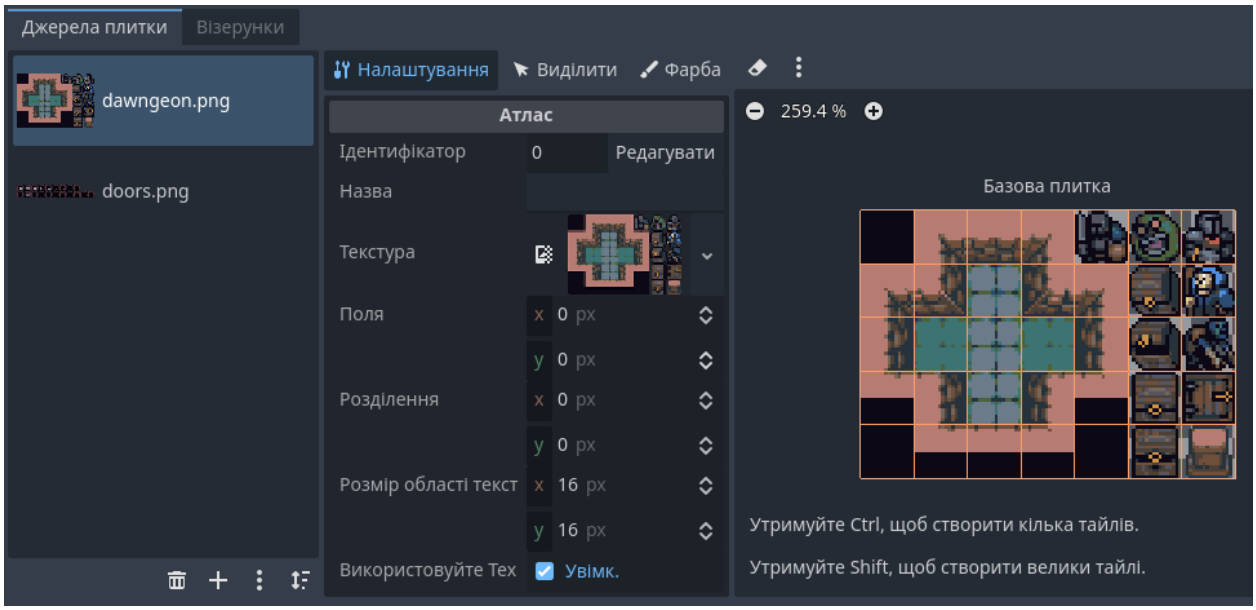

Рис. 3.8. Вкладка «TileSet»

Переходимо до наступної вкладки. У ній ми можемо малювати наш ігровий простір. Для цього у інструмента обираємо олівець, натискаємо на потрібну текстуру та відображаємо їх на робочій області. Для того, щоб видалити непотрібні текстури з робочої області, переключаємо інструмент на гумку. Так як ми вказали розмірність нашого вікна, у робочій області відображаються границі видимості екрану. Заповнюємо його текстурами (рис. 3.9).

На цьому етапі ми маємо сцену, в якій відображається кімната на увесь екран. Для перевірки будь-якої сцени, запускаємо її, натиснувши кнопку на панелі відтворення. По стандарту камеру буде центруватись на середині робочої області.

Наступним кроком створимо персонажа з певними анімаціями, щоб відобразити його певні ігрові стани, такі як рух, стояння, тощо.

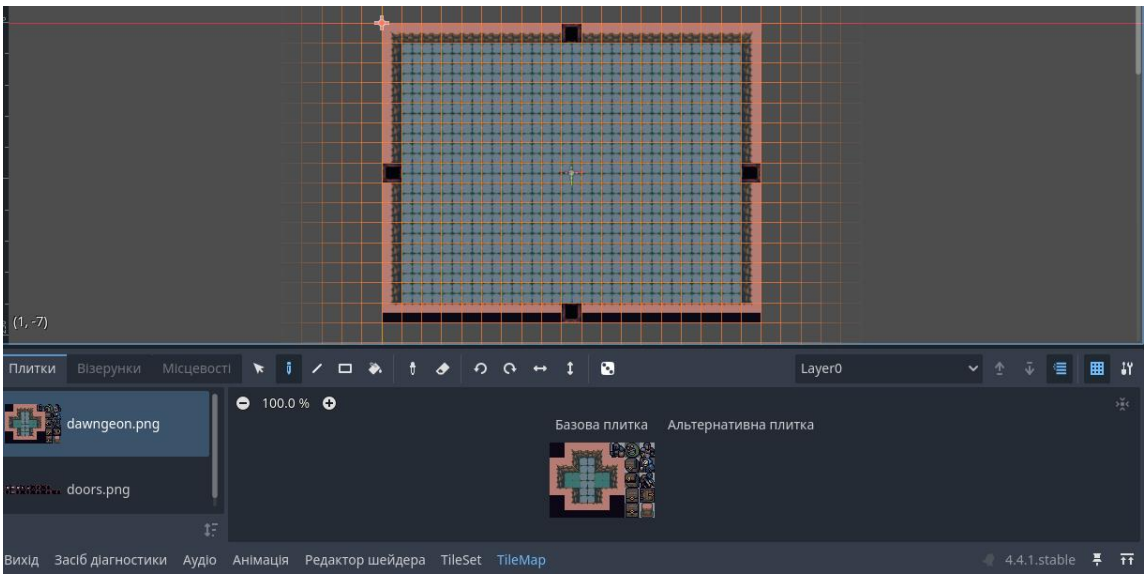

Рис. 3.9. Створення ігрового простору

Для цього створюємо нову сцену, додаємо вузол «AnimatedSprite2D», який буде відповідати за відображення колекції анімацій персонажа. Після створення вузла, обираємо його та переходимо до інспектора. У вкладці «Animation – Sprite Frames», обираємо створити новий. Це потрібно для того, щоб створити новий елемент для завантаження колекції зображень для виконання анімації на їх основі (рис. 3.10).

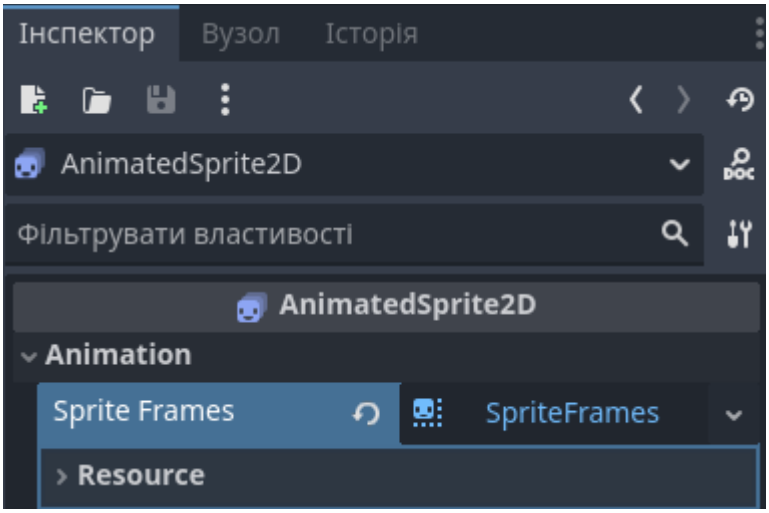

Рис. 3.10. Інспектор

Після створення, у нижній частині екрану відкриється вікно для створення анімацій. У лівій частині ми обираємо швидкість анімації та створюємо шаблони з кодовими назвами. З правої частини – робоча область, у яку ми завантажуємо зображення, які ми хочемо поєднати у анімацію (рис. 3.11).

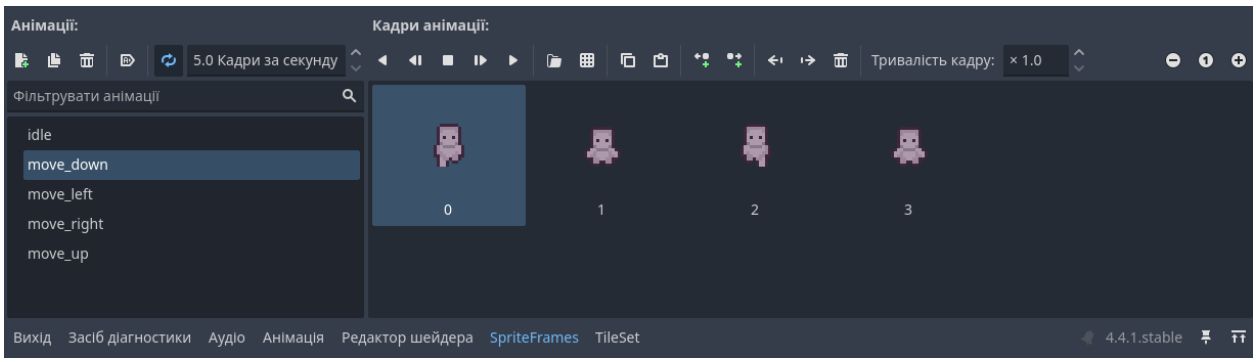

Рис. 3.11. Редактор анімації

Створюємо шаблони. Додаємо до них наші зображення за допомогою кнопок додавання кадрів (якщо у нас один кадр, обираємо значок папки або сусідній, якщо набір кадрів знаходиться на одному зображенні), налаштовуємо їх, обираємо правильну послідовність, обов'язково натискаємо кнопку циклічності, щоб анімація повторно програвалась при закінченні. Для перевірки анімації можемо її запустити використовуючи відповідні кнопки інтерфейсу редактора.

Також можемо одразу додати посох нашому персонажу. Для цього треба створити наступну конструкцію вузлів: «Node2D», у якій знаходиться «Sprite2D» (рис. 3.12).

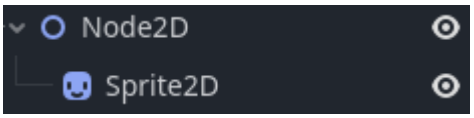

Рис. 3.13. Структура вузлів для додавання зображення

Переходимо до інспектора, та додаємо зображення моделі посоху. Анімацію посоху буде реалізовано пізніше, на етапі створення коду. А поки розміщуємо зброю на потрібному місці біля персонажу (рис. 3.14).

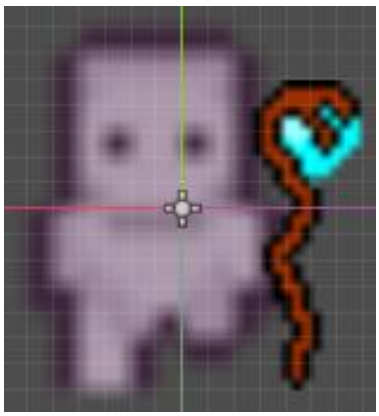

Рис. 3.14. Зображення персонажу

На цьому етапі ми вже маємо готовий шаблон кімнат та персонажу. Аналогічно до процесу створення персонажа, додаємо нові сцени для снарядів, ворогів, босів тощо (рис. 3.15). Створюємо анімації та зберігаємо у відповідних папках.

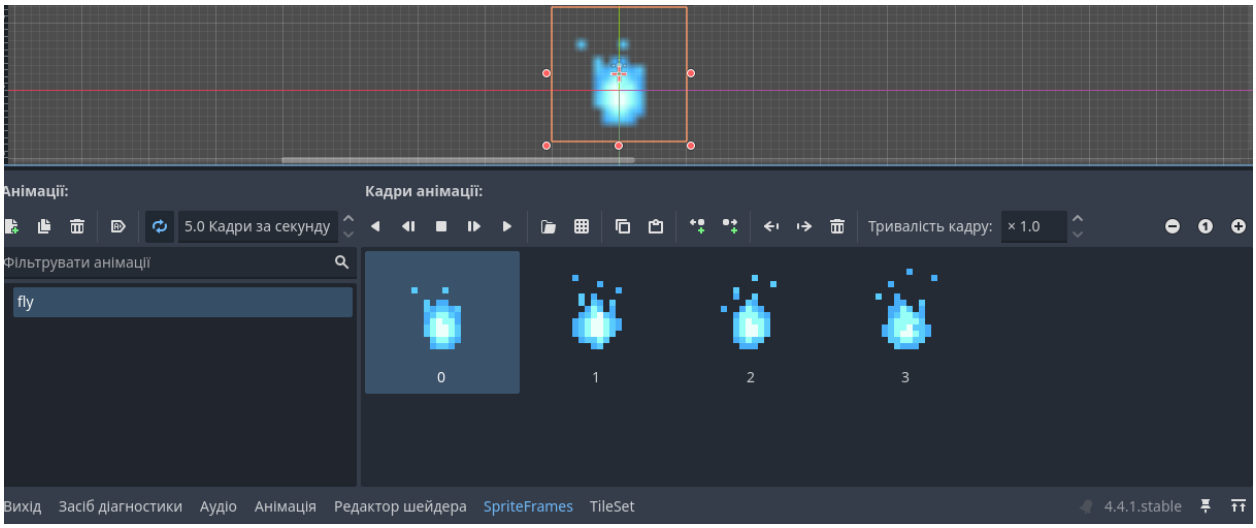

Рис. 3.15. Приклад анімованого снаряду персонажа

Наступним кроком створимо шаблон інтерфейсу користувача. Для цього створюємо нову сцену з назвою «HUD» та додаємо у неї елементи, які будуть відображатись на екрані.

Розпочнемо з кількості здоров’я гравця. Розташовуємо вузол типу «HBoxContainer», який буде містити 6 вузлів типу «TextureRect». Контейнер буде визначати розташування поля, де буде відображатись поточне здоров'я, а за допомогою текстур ми зможемо відобразити потрібну нам кількість здоров’я, враховуючи максимальне здоров’я та можливість збільшення кількості його контейнерів до 6.

Далі розташуємо кількість монет та ключів гравця. Для цього скористаємось вузлами типу «AnimatedSprite2D» для анімації зображень та «Label» для відображення кількості розхідників користувача. Створення анімації відбувається аналогічно до минулих кроків.

Останнім чином створюємо зону під мапу. Для цього додаємо вузол типу «Control». Поки що він не буде нічого відображати, але у режимі редактора ми зможемо побачити зону його відображення на екрані.

Додавши усі елементи ми маємо окрему сцену, яка відповідає за

відображення інтерфейсу користувача, який ми можемо зручно змінювати та оновлювати за власним бажанням (рис. 3.16).

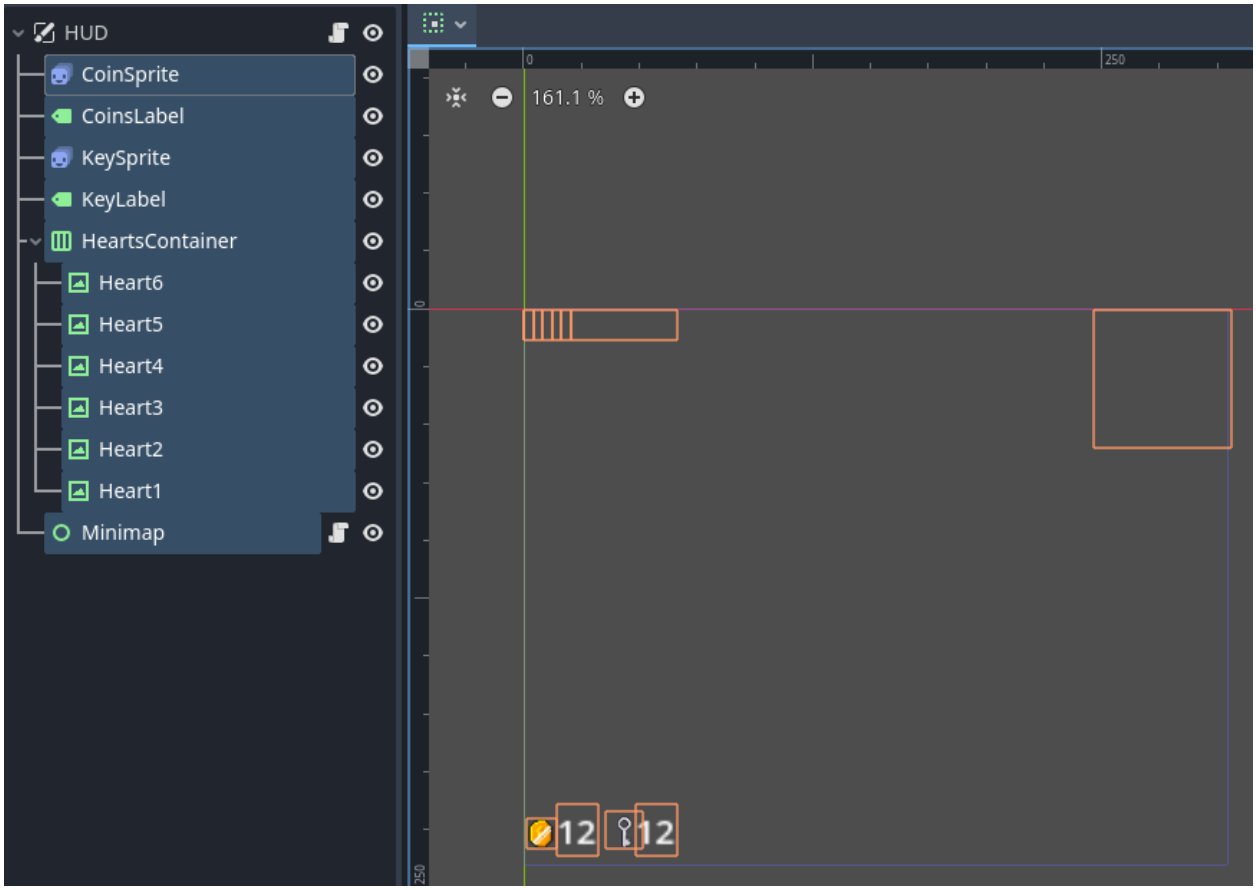

Рис. 3.16. Створення користувацького інтерфейсу

3.3. Розробка ключових механік

Маючи усі потрібні нам графічні елементи, залишилось лише поєднати їх зі скриптами, які будуть задавати логіку взаємодії вузлів та сцен.

Розпочнемо з персонажу: створимо логіку пересування персонажа та пострілів за клавіш клавіатури, колізії та коректного відображення посоху. Для цього повертаємось у сцену Player та створюємо необхідні для цього вузли типу «FiniteStateMachine» для механізму пересування та «CollisionShape2D» для колізії. Створивши елемент колізії, розташовуємо її на персонажі, обираємо головний вузол та в інспекторі у вкладці «Collision» визначаємо правила взаємодії. Як ми раніше визначали у налаштуваннях, 2 шар відповідає за персонажа, а 1 та 3 за світ та ворогів аналогічно. Отже за таким принципом у першому полі «Layer» визначаємо шар обраного об’єкта, а у полі «Mask» з чим він зможе взаємодіяти (рис. 3.17). Аналогічні дії виконуємо для усіх інших

персонажів, ворогів, снарядів, тощо.

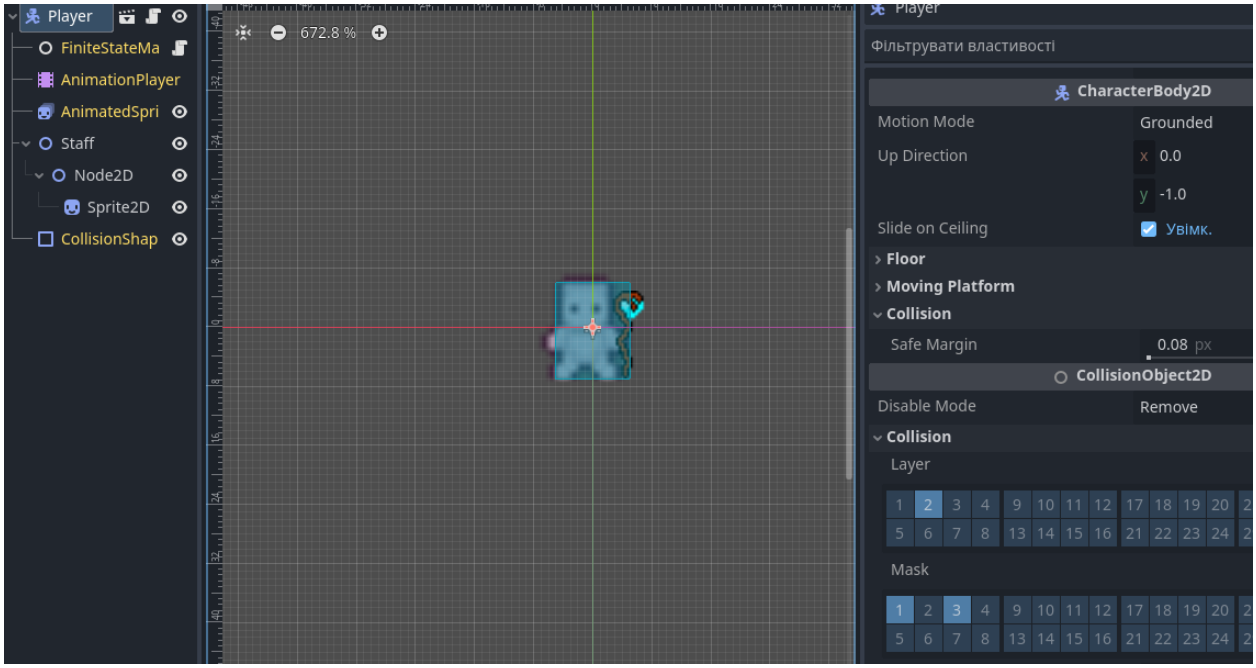

Рис. 3.17. Оновлена сцена «Player»

Далі створюємо скрипти, які потрібні для реалізації логіки персонажа. Для цього натискаємо на відповідну кнопку біля вузла. На початку коду визначаємо наслідування класу та його назву. Це потрібно для того, щоб скрипт розумів, з якою частиною спеціальних функцій він буде працювати та до якого об’єкту звертатись. Далі за допомогою конструкції «@export var» визначаємо параметри персонажа, такі як: сила атаки, швидкість снарядів, персонажа та його пришвидшення, частоту пострілів, максимальне та поточне здоров’я, поверх та кількість монет чи ключів. Після цього ми отримуємо дані з інших сцен, для того, щоб змінювати текстові поля у інтерфейсі користувача, редагувати положення посоху на сцені та використовувати різні анімації персонажа, як ось рух у різні сторони. Після цього визначаємо інші змінні та константи, які потрібні для реалізації механік. У прикладі визначено: стала коефіцієнту тертя, змінні можливості стріляти, положення посоху, його локальної анімації та вектору переміщення персонажа. Також нам потрібно завантажити сцену снаряду, для того щоб використовувати її, адже у цьому файлі ми маємо також і його змінні (рис. 3.18).

```
1   extends CharacterBody2D
2   class_name Character
3
4   @export var damage: int = 1
5   @export var fire_rate: float = 0.5
6   @export var projectile_speed: float = 150.0
7   @export var move_speed: float = 80
8   @export var max_health: int = 5
9   @export var current_hearts: int = 4
10  @export var current_floor: int = 1
11  @export var coins: int = 14
12  @export var keys: int = 14
13  @export var acceleration: int = 40
14
15  @onready var hud = get_parent().get_node("HUD")
16  @onready var staff: Node2D = $Staff
17  @onready var animated_sprite: AnimatedSprite2D = get_node("AnimatedSprite2D")
18
19  var projectile_scene = preload("res://Characters/Projectiles/Projectile.tscn")
20  var can_shoot := true
21  const FRICTION: float = 0.15
22  var staff_base_position := Vector2(0, 0)
23  var bob_offset := 0
24  var mov_direction: Vector2 = Vector2.ZERO
```

Рис. 3.18. Змінні персонажу та імпорт потрібних елементів

Після визначення усіх потрібних змінних, перейдемо до створення функцій. Першою з них є переміщення персонажу. Спочатку перевіряємо, чи рухається наш персонаж. Якщо вектор руху не дорівнює нулю, виконуємо наступні дії: нормалізуємо вектор, щоб уникнути зайвої швидкості при руху по діагоналі, далі розраховуємо швидкість персонажу у моменті, помноживши напрямок руху на пришвидшення, а також стабілізуємо значення до максимальної швидкості. У результаті персонаж правильно переміщається у 8 сторін з однаковою, максимальною швидкістю (рис. 3.19).

```
func move() -> void:
    if mov_direction != Vector2.ZERO:
        mov_direction = mov_direction.normalized()
        velocity += mov_direction * acceleration
        velocity = velocity.limit_length(move_speed)
```

Рис. 3.19. Функція «move»

Наступна функція зчитує введення з клавіатури. Як ми і визначали у налаштуваннях раніше, клавіши мають певні назви. При введенні, скрипт обробляє натисненні клавіши, та на основі їх основі змінює локальну змінну «input_vector», щоб потім нормалізувати напрямок та передати її у глобальну змінну на обробку іншими функціями (рис. 3.20).

```
func get_input() -> void:
    var input_vector = Vector2.ZERO
    if Input.is_action_pressed("move_right"):
        input_vector.x += 1
    if Input.is_action_pressed("move_left"):
        input_vector.x -= 1
    if Input.is_action_pressed("move_down"):
        input_vector.y += 1
    if Input.is_action_pressed("move_up"):
        input_vector.y -= 1
    mov_direction = input_vector.normalized()
```

Рис. 3.20. Функція get_input()

Тепер наш персонаж рухається у 8 напрямків за допомогою натискання на клавіши «WASD». Для того, щоб додати анімацію пересування персонажу, використаємо наш вхідний вектор. За допомогою умовно-логічних конструкцій визначаємо напрямок та наявність руху персонажа, щоб відтворити потрібну анімацію, за допомогою методу «.play» (рис. 3.21).

```
if input_vector != Vector2.ZERO:
    if abs(input_vector.x) > abs(input_vector.y):
        sprite.play("move_right" if input_vector.x > 0 else "move_left")
    else:
        sprite.play("move_down" if input_vector.y > 0 else "move_up")
else:
    sprite.play("idle")
```

Рис. 3.21. Виклик анімації персонажа

Далі опишемо функції для логіки посоху та його коректного відображення на екрані. Спочатку за допомогою вбудованої функції _ready() підключаємо обробник подій для зміни кадрів та встановлюємо початкове

положення посоху (рис. 3.22).

```
func _ready():
    animated_sprite.frame_changed.connect(_on_frame_changed)
    staff.position = staff_base_position
```

Рис. 3.22. Функція _ready()

Далі, описуємо анімацію гойдання посоху. Для цього змінюємо його положення на 2 пікселі з кожним оновленням посоху. Після цього повертаємо значення до параметру позиції посоху (рис. 3.23).

```
func _on_frame_changed():
    bob_offset = -2 if bob_offset == 0 else 0
    staff.position = staff_base_position + Vector2(0, bob_offset)
```

Рис. 3.23. Функція _jn_frame_changed()

Коли наш персонаж рухається, його тіло повернуте у обрану сторону, отже й посох, при русі уверх та вліво має змінювати руку персонажа. Для реалізації цієї логіки необхідно віддзеркалити положення посоху відповідно до центру персонажа. Якщо наш персонаж стоїть або рухається вправо чи вниз, ми повертаємо звичайне значення за допомогою конструкції «$Staff.scale.x», що змінює параметри відображення вузла з посохом. Якщо функція помічає рух в інші сторони, вона повертає -1, що віддзеркалює посох відповідно осі Y (рис. 3.24).

```
func update_staff_direction():
    if mov_direction == Vector2.ZERO:
        $Staff.scale.x = 1
    else:
        var dir = mov_direction.normalized()

        if dir.x < 0 or dir.y < 0:
            $Staff.scale.x = -1
            staff_base_position.x = -1
        else:
            $Staff.scale.x = 1
```

Рис. 3.24. Функція update_staff_direction()

Залишилось навчити нашого персонажа стріляти у визначену сторону.

На початку, якщо нашому персонажу не дозволено стріляти (наприклад зовнішні ефекти або перезарядка пострілу), ми повертаємось з функції. Далі зчитуємо відповідні натискання клавіш з клавіатури за допомогою методу «.is_action_pressed()» та записуємо значення у відповідну змінну. Далі обробляємо отриману інформацію: якщо визначено вектор (напрямок стрільби) та сцену зі снарядом, то створюємо на екрані снаряд, якому передаємо характеристики. Після цього за допомогою методу «.add_child()» створюємо снаряд у дереві нашої сцени. Це потрібно для того, щоб відображати на екрані більше одного снаряду за раз. Потім за допомогою методу «.timeout» та оператору «await», який виконується асинхронно, визначаємо час, у який наш персонаж не зможе стріляти (рис. 3.25).

```
func shoot():
    if not can_shoot:
        return
    var dir = Vector2.ZERO
    if Input.is_action_pressed("attack_up"):
        dir = Vector2.UP
    elif Input.is_action_pressed("attack_down"):
        dir = Vector2.DOWN
    elif Input.is_action_pressed("attack_left"):
        dir = Vector2.LEFT
    elif Input.is_action_pressed("attack_right"):
        dir = Vector2.RIGHT
    if dir != Vector2.ZERO and projectile_scene:
        var proj = projectile_scene.instantiate()
        proj.position = global_position + dir * 10
        proj.direction = dir
        proj.speed = projectile_speed
        proj.damage = damage
        get_tree().current_scene.add_child(proj)
        can_shoot = false
        await get_tree().create_timer(fire_rate).timeout
        can_shoot = true
```

Рис. 3.25. Функція shoot()

Усі вищеописані функції постійно викликаються за допомогою

вбудованої функції _physics_process(). Так як ми зберігаємо ключові змінні значення здоров'я, монет, ключів у цьому скрипті, то нам потрібно постійно звертатись до функцій скрипту сцени «HUD». У них ми передаємо ці значення на подальшу обробку. Також, для зупинення персонажу, за допомогою функції «lerp», ми зменшуємо швидкість персонажу, в залежності від прискорення та направлення руху (рис. 3.26).

```
func _physics_process(_delta: float) -> void:
    move()
    get_input()
    shoot()
    move_and_slide()
    hud.update_health(current_hearts,max_health)
    hud.update_hud(coins,keys)
    update_staff_direction()
    velocity = lerp(velocity, Vector2.ZERO, FRICTION)
```

Рис. 3.26. Функція _physics_process()

Раніше, ми створили вузол типу «FiniteStateMachine», який повинен відповідати за коректний перехід між анімаціями та більш детально обробляти стани її логіки. Створимо окремий скрипт для нього, натиснувши на відповідну кнопку біля вузла у інтерфейсі дерева вузлів.

Аналогічно до минулого, створюємо звернення з класом вузла. Також оголосимо змінні для індексації попереднього та поточного стану та список усіх можливих індексів. Приєднуємо їх до персонажу у змінну parent функцією get_parent() та створюємо об'єкт animation_player, що буде відповідати за запуск анімації (рис. 3.27).

```
extends Node
class_name FiniteStateMachine

var states: Dictionary = {}
var previous_state: int = -1
var current_state: int = -1

@onready var parent: Character = get_parent() as Character
@onready var animation_player: AnimationPlayer = parent.get_node("AnimationPlayer")
```

Рис. 3.27. Необхідні змінні та імпорт потрібних елементів

Основна логіка реалізована у функції _physics_process(), яка кожен кадр перевіряє логіку поведінки персонажу, потім аналізує потребу у переході до іншого стану і якщо потреба є, виконує перехід до потрібної анімації (рис. 3.28).

```
func _physics_process(delta: float) -> void:
    if current_state != -1:
        _state_logic(delta)
        var transition: int = _get_transition()
        if transition != -1:
            set_state(transition)
```

Рис. 3.28. Функція _physics_process()

Функція _state_logic() зчитує пересування та ввід користувача за допомогою звертання до функції попереднього скрипту, який ми визначили раніше (рис. 3.29)

```
func _state_logic(_delta: float) -> void:
    parent.get_input()
    parent.move()
```

Рис. 3.29. Функція _state_logic()

Маючи потрібні дані, ми можемо виконати обчислення для визначення нового стану. Якщо швидкість персонажу дуже низька, будемо вважати, що він не рухається. У іншому випадку, ми нормалізуємо вектор руху та визначаємо, яку анімацію виконати (рис. 3.30).

```
func _get_transition() -> int:
    if parent.velocity.length() < 10:
        return states.idle
    var dir = parent.velocity.normalized()
    if abs(dir.x) > abs(dir.y):
        return states.move_right if dir.x > 0 else states.move_left
    else:
        return states.move_down if dir.y > 0 else states.move_up
```

Рис. 3.30. Функція _get_transition()

Функція _add_state() додає у наш список усі анімації, що присутні у вузлі

анімації (рис. 3.31).

```
func _add_state(new_state: String) -> void:
    states[new_state] = states.size()
```

Рис. 3.31. Функція _add_state()

Для зміни стану, тобто переходу до нового стану, використовується функція set_state(), яка виходить з минулого стану, оновлює дані значень станів та встановлює новий, актуальний стан (рис. 3.32).

```
func set_state(new_state: int) -> void:
    _exit_state(current_state)
    previous_state = current_state
    current_state = new_state
    _enter_state(previous_state, current_state)
```

Рис. 3.32. Функція set_state()

Остання функція цього файлу – _enter_state(). Вона відповідає за переходи після визначення стану, порівнюючи вхідне значення назви анімації, а також за допомогою методу .play() відтворює її (рис.3.33).

```
func _enter_state(_previous_state: int, _new_state: int) -> void:
    match _new_state:
        states.idle:
            animation_player.play("idle")
        states.move_up:
            animation_player.play("walk_up")
        states.move_down:
            animation_player.play("walk_down")
        states.move_left:
            animation_player.play("walk_left")
        states.move_right:
            animation_player.play("walk_right")
```

Рис. 3.33. Функція _enter_state()

Для відображення інтерфейсу, перейдемо до сцени «HUD», у якій для основного вузла створимо скрипт «HUD.gd». На його початку визначаємо тип вузла, а саме «CanvasLayer». Відвантажуємо у змінні усі вузли, які ми хочемо редагувати. У нашому прикладі ми маємо: кількість монет та ключів, та їх

спрайти з анімаціями та мапу. Також визначаємо масив контейнерів для сердець та зображення станів сердець (рис. 3.34).

```
extends CanvasLayer

@onready var coins_label: Label = $CoinsLabel
@onready var key_label: Label = $KeysLabel
@onready var coin_sprite: AnimatedSprite2D = $CoinSprite
@onready var key_sprite: AnimatedSprite2D = $KeySprite
@onready var minimap: Control = $Minimap

@onready var hearts = [
    $HeartsContainer/Heart6,
    $HeartsContainer/Heart5,
    $HeartsContainer/Heart4,
    $HeartsContainer/Heart3,
    $HeartsContainer/Heart2,
    $HeartsContainer/Heart1,
]
var full_heart = preload("res://Assets/UI/heart_full.png")
var half_heart = preload("res://Assets/UI/heart_half.png")
var empty_heart = preload("res://Assets/UI/heart_empty.png")
```

Рис. 3.34. Необхідні змінні та імпорт потрібних елементів

У основній функції _ready(), вмикаємо анімацію елементам інтерфейсу а також встановлюємо видимість для мапи, звернувшись до параметру «.visible» вузла (рис.3.35).

```
func _ready():
    coin_sprite.play("coin")
    key_sprite.play("key")
    minimap.visible = true
```

Рис. 3.35. Функція _ready()

Для оновлення мапи, використовується функція update_minimap(), яка звертається до функції скрипту мапи (рис. 3.36)

```
func update_minimap(used_positions: Dictionary):
    if minimap:
        minimap.draw_rooms(used_positions)
```

Рис. 3.36. Функція update_minimap()

Для оновлення здоров’я звертаємось до функції update_health(), яка виконує обробку коректного відображення здоров’я на екрані. На початку вона аналізує максимальну кількість контейнерів, розраховує значення поточного контейнера, відображає його та переходить до іншого (рис. 3.37).

```
func update_health(current_hp: int, max_hp: int):
    for i in range(max_hp):
        var heart_value = clamp(current_hp - i * 2, 0, 2)
        match heart_value:
            2:
                hearts[i].texture = full_heart
            1:
                hearts[i].texture = half_heart
            _:
                hearts[i].texture = empty_heart
```

Рис. 3.37. Функція update_health()

Остання функція користувацького інтерфейсу (_update_hud()), відновлює значення монет та ключів, змінюючи аналогічні вузли типу «Label» (рис. 3.38)

```
func update_hud( coins: int, keys: int) -> void:
    coins_label.text = "%d" % coins
    key_label.text = "%d" % keys
```

Рис. 3.38. Функція _update_hud()

Перейдемо реалізації процедурної генерації та механіки переходу між кімнатами. Для цього створимо нову сцену, яка буде виконувати ці дії. Створимо додатковий вузол типу «Camera2D», для позначення центру кімнати. Далі нам треба створити скрипт головного вузла.

В ньому на початку приймаємо сцени, які є шаблонами кімнат різних типів та масив зі значеннями видів кімнат. Це треба для того, щоб із певної кількості сцен випадковим чином обрати шаблони кімнат. Далі створюємо змінні для керування генерацією, а саме кількістю кімнат поверху та їх типів. Після цього отримуємо значення камери, гравця та мапи. Для камери потрібно визначити кімнату, позицію, список позицій, які вона може займати та розміри

кімнати та напрямки, для правильного переходу між ними (рис. 3.39).

```
extends Node2D

enum RoomType {
    SPAWN,
    NORMAL,
    BOSS,
    SHOP,
    ITEM
}

@export var normal_room_scenes: Array[PackedScene]
@export var boss_room_scenes: Array[PackedScene]
@export var shop_room_scenes: Array[PackedScene]
@export var item_room_scenes: Array[PackedScene]
@export var spawn_room_scene: PackedScene
@export var current_floor: int = 1
@export var room_scene: PackedScene
@export var room_count: int = 8
@export var boss_room_count: int = 1
@export var shop_room_count: int = 1
@export var item_room_count: int = 1
@onready var camera: Camera2D = $RoomCamera
@export var minimap: Control
@export var player: Node2D
var camera_target_position: Vector2
var camera_speed := 5.0
var used_positions := {}
var room_size := Vector2(304, 240)
var directions = [
    Vector2.LEFT, Vector2.RIGHT, Vector2.UP, Vector2.DOWN
]
```

Рис. 3.39. Необхідні змінні та імпорт потрібних елементів

Основна функція _ready() викликає ініціалізацію мапи, генерацію структури рівня, заповнення її кімнатами та розташуванням дверей. В залежності від поверху змінюється кількість звичайних кімнат для генерації (рис. 3.40).

```
func _ready():
    call_deferred("_init_minimap")
    match current_floor:
        1:
            generate_floor(10, 1, 1, 1)
            spawn_all_rooms()
            process_doors_and_walls()
        2: generate_floor(12, 1, 1, 1)
        3: generate_floor(16, 1, 1, 1)
```

Рис. 3.40. Функція _ready()

Генератор рівнів створює сітку, яку починає з координат (0;0) та поступово заповнює її кімнатами. На початку відбувається очищення масиву використаних кімнат. Дале створюється початкове положення (центр поверху), який спочатку заповнюється звичайними кімнатами. Далі виконується цикл, котрий додає кімнати до зазначеного моменту (кількості кімнат), для цього вона звертається до функції для отримання доступного напрямку, і якщо отримує позитивну відповідь, додає кімнату у масив використаних кімнат. Якщо зайнято, випадковим чином знову обирається напрямок та кімната, з якої продовжиться генерація. Після цього на готовому шаблоні звичайних кімнат, за зазначеними правилами, додаються додаткові кімнати, такі як: кімната з предметом, боссом та магазин. У кінці генерується початкова кімната, яка заміняє першу створену звичайну кімнату.

```
func generate_floor(normal_room_count: int,boss_count: int, shop_count: int, item_count: int):
    used_positions.clear()
    var start_pos := Vector2.ZERO
    used_positions[start_pos] = "Normal Room"
    var path := [start_pos]
    while used_positions.size() < normal_room_count:
        var current_pos = path[randi() % path.size()]
        var available_dirs = _get_available_directions(current_pos)
        if available_dirs.is_empty():
            continue
        var dir = available_dirs.pick_random()
        var new_pos = current_pos + dir
        used_positions[new_pos] = "Normal Room"
        path.append(new_pos)
    print("Сгенеровано кімнат: %d" % used_positions.size())
    add_boss_room()
    add_special_rooms(shop_count, "Shop Room")
    add_special_rooms(item_count, "Item Room")
    if used_positions.has(start_pos):
        used_positions[start_pos] = "Spawn Room"
```

Рис. 3.41. Функція generate_floor()

Для отримання доступності кімнат для розміщення по сусідству, використовується функція _get_available_directions(), яка перебирає усі положення кімнат, та визначає, чи є вільні кандидати у 4 сторонах, по сусідству з обраною. Результатом виконання є масив з напрямками, які вільні

для генерації кімнат (рис. 3.42).

```
func _get_available_directions(pos: Vector2) -> Array:
    var dirs := []
    for dir in directions:
        var neighbor = pos + dir
        if not used_positions.has(neighbor):
            dirs.append(dir)
    return dirs
```

Рис. 3.42. Функція _get_available_directions()

Усі додаткові кімнати генеруються окремо, не заміняючи звичайні. Кімната з босом генерується, як найбільш далека кімната від початкової кімнати. Для початку, звичайні кімнати переносяться у масив, для визначення найдальшої кімнати. Потім в залежності від цього, створюється додаткова кімната, яка отримає відповідне маркування (рис. 3.43).

```
func add_boss_room():
    var start_pos = Vector2.ZERO
    var normal_rooms = []
    for pos in used_positions.keys():
        if used_positions[pos] == "Normal Room":
            normal_rooms.append(pos)
    normal_rooms.sort_custom(func(a, b):
        return a.distance_squared_to(start_pos) > b.distance_squared_to(start_pos)
    )
    for base_pos in normal_rooms:
        for dir in directions:
            var boss_pos = base_pos + dir
            if used_positions.has(boss_pos):
                continue
            var neighbor_count = 0
            for neighbor_dir in directions:
                if used_positions.has(boss_pos + neighbor_dir):
                    neighbor_count += 1
            if neighbor_count == 1:
                used_positions[boss_pos] = "Boss Room"
                print("✅ Позиція босс-руми: ", boss_pos)
                return
    print("⚠ Не вдалось додати босс-рум!")
```

Рис. 3.43. Функція add_boss_room()

Так як правила генерації особливих кімнат (магазин та з предметом) однакові, тобто вони не мають мати сусіда, тобто лише 1 вихід до звичайної кімнати, їх генерація об’єднана в одну функцію. Принцип генерації схожий до минулого. Створюється список кандидатів, у який входять кімнати з 1 допустимим сусідом. Потім кандидати перемішуються, та у випадкових місцях з’являються 2 кімнати (рис. 3.44).

```
func add_special_rooms(count: int, room_type: String):
    var candidates = []
    for base_pos in used_positions.keys():
        if used_positions[base_pos] != "Normal Room":
            continue
        for dir in directions:
            var new_pos = base_pos + dir
            if used_positions.has(new_pos):
                continue
            var neighbor_count = 0
            for neighbor_dir in directions:
                if used_positions.has(new_pos + neighbor_dir):
                    neighbor_count += 1
            if neighbor_count == 1:
                candidates.append(new_pos)
    candidates.shuffle()
    var placed = 0
    for pos in candidates:
        if placed >= count:
            break
        used_positions[pos] = room_type
        placed += 1
        print("✅ %s додана в позицію: %s" % [room_type, str(pos)])
    if placed < count:
        print("⚠ Не вдалось додати усі %s кімнати! (%d из %d)" % [room_type, placed, count])
```

Рис. 3.44. Функція add_special_rooms()

Коли сітка кімнат була створена, потрібно заповнити цю сітку фізично кімнатами. Для цього перебираємо усі кімнати з масиву зайнятих позицій. Для кожної кімнати перевіряємо її тип, і на місці її координат, створюємо кімнату потрібного типу проте випадкового шаблону, визначеному на початку. Для стартової кімнати, варіацій не існує, тому вона завжди буде однаковою на будь-якому поверсі (рис. 3.45).

```
func spawn_all_rooms():
    for grid_pos in used_positions.keys():
        var type_str = used_positions[grid_pos]
        var room_instance
        match type_str:
            "Boss Room":
                room_instance = boss_room_scenes.pick_random().instantiate()
            "Shop Room":
                room_instance = shop_room_scenes.pick_random().instantiate()
            "Item Room":
                room_instance = item_room_scenes.pick_random().instantiate()
            "Spawn Room":
                room_instance = spawn_room_scene.instantiate()
            "Normal Room":
                room_instance = normal_room_scenes.pick_random().instantiate()
            _:
                continue

        add_child(room_instance)
        room_instance.position = grid_pos * room_size
```

Рис. 3.45. Функція spawn_all_rooms()

Для реалізації працездатності камери та оновлення мапи, використовується функція _process(). Вона відслідковує положення персонажа та реагує на торкання границь кімнати для переходу. Після цього, камера плавно переміщується на одну кімнату у потрібну сторону та відновлюється поточне положення персонажа на мапі (рис. 3.46).

```
func _process(delta):
    if player == null:
        return
    var player_room_grid = Vector2(
        floor(player.position.x / room_size.x),
        floor(player.position.y / room_size.y)
    )
    camera_target_position = (player_room_grid * room_size) + (room_size / 2.0)
    if camera.position != camera_target_position:
        camera.position = camera.position.lerp(camera_target_position, delta * camera_speed)
    if minimap:
        minimap.set_current_room(player_room_grid)
```

Рис. 3.46. Функція _process()

Остання функція виконує декілька важливих задач. Основним є заміна усіх тайлів дверей на стіни, якщо немає сусіда. Для цього, на початку, зазначено координати тайлів дверей у кімнаті та координат тайлів стін на мапі

тайлів кімнат. Далі, програма знову аналізує сусідство кімнат та заміняє двері на стіни у випадку сусідства. Також на цьому етапі створюється мапа, використовуючи оброблені дані, які збережені у змінну «used_positions» (рис. 3.47).

```
func process_doors_and_walls():
    var door_data = {
        Vector2.UP:    { "cell": Vector2i(9, 0),  "tile": Vector2i(2, 0) },
        Vector2.RIGHT: { "cell": Vector2i(18, 7), "tile": Vector2i(4, 2) },
        Vector2.DOWN:  { "cell": Vector2i(9, 14), "tile": Vector2i(2, 4) },
        Vector2.LEFT:  { "cell": Vector2i(0, 7),  "tile": Vector2i(0, 2) }
    }
    for room_node in get_children():
        if not room_node is Node2D:
            continue
        var grid_pos = room_node.position / room_size
        var tilemap = room_node.get_node_or_null("RoomTilemap")
        if tilemap == null:
            print("Не знайден TileMap в кімнаті", room_node.name)
            continue
        print("Кімната в позиції ", grid_pos, " перевірка дверей:")
        for dir in directions:
            var neighbor_pos = grid_pos + dir
            var door = door_data[dir]
            var door_cell = door["cell"]
            var wall_tile_coords = door["tile"]
            if not used_positions.has(neighbor_pos):
                print("  ❌ Не існує сусіда за напрямком ", dir, " — заміняється двері на стіну.")
                tilemap.set_cell(0, door_cell, 0, wall_tile_coords)
            else:
                print("  ✅ Сусід існує за напрямком ", dir)
    var hud = get_node("/root/Game/HUD")
    minimap = hud.minimap
    if minimap:
        print(used_positions)
        minimap.update_minimap(used_positions)
```

Рис. 3.47. Функція process_doors_and_walls()

Для відображення кімнат на карті, повернемось до сцени «HUD» та створимо на місці вузлу типу «Control» скрипт для мапи. На початку визначаємо розмірність сітки кімнат та змінні для збереження типів кімнат та позиції гравця. Функція update_minimap(), очищує дані про кімнати, оновлює значення положень об’єктів та перемальовує мапу для коректного відображення (рис. 3.48).

```
extends Control

const ROOM_ICON_SIZE := Vector2(8, 8)
var room_data: Dictionary = {}
var current_room_pos: Vector2 = Vector2.ZERO
var offset := Vector2.ZERO

func update_minimap(room_positions: Dictionary):
    room_data.clear()
    for grid_pos in room_positions.keys():
        var type_str := str(room_positions[grid_pos])
        room_data[grid_pos] = type_str
    _update_offset()
    queue_redraw()
```

Рис. 3.48. Необхідні змінні та функція update_minimap()

Функція _update_offset() визначає мінімальні координати розміщення кімнат та на їх основі зсуває мапу для розташування по середині. А функція set_current_room() допомагає визначити, у якій позиції гравець зараз знаходиться відповідно до мапи (рис. 3.49).

```
func _update_offset():
    if room_data.is_empty():
        offset = Vector2.ZERO
        return
    var min_x = room_data.keys()[0].x
    var min_y = room_data.keys()[0].y
    for pos in room_data.keys():
        min_x = min(min_x, pos.x)
        min_y = min(min_y, pos.y)
    offset = Vector2(min_x, min_y)

func set_current_room(pos: Vector2):
    current_room_pos = pos
    queue_redraw()
```

Рис. 3.49. Функції _update_offset() та set_current_room()

Останні 2 функції відповідають за створення прямокутників у вигляді

кімнат, та їх розташуванням відповідно до розміщення кімнат після генерації, а також визначення кольорів кімнат, в залежності від їх типу. Так як початкова кімната не зазначається, то вона буде намальована у сірий колір (рис. 3.50).

```
func _draw():
    for grid_pos in room_data.keys():
        var type = room_data[grid_pos]
        var color := _get_room_color(type)
        var draw_pos = (grid_pos - offset) * ROOM_ICON_SIZE
        draw_rect(Rect2(draw_pos, ROOM_ICON_SIZE), color)
    var current_draw_pos = (current_room_pos - offset) * ROOM_ICON_SIZE
    draw_rect(Rect2(current_draw_pos, ROOM_ICON_SIZE), Color.CYAN, false)

func _get_room_color(type: String) -> Color:
    match type:
        "Normal Room": return Color.WHITE
        "Boss Room": return Color.RED
        "Shop Room": return Color.YELLOW
        "Item Room": return Color.LIME_GREEN
        _: return Color.GRAY
```

Рис. 3.50. Функції _draw() та _get_room_color()

Таким чином ми маємо повністю справну генерацію рівнів з правилами генерування особливих кімнат, її відображення на мапі, персонажа, який вміє рухатись та випускати снаряди, переходи між кімнатами а також повноцінний користувацький інтерфейс.

3.4. Тестування та оптимізація

Для тестування гри можна використовувати поєднання декількох сцен на одній. Така дія використовується для того, щоб оцінити, як гра поводить себе у різноманітній комбінації поєднання сцен та вузлів. На етапі створення персонажа, його було поміщено у одну з кімнат. Під чай тестування, можна відкоригувати усі початкові змінні, такі як швидкість, пришвидшення, сповільнення тощо. Також відбувалась перевірка коректно8го відображення анімацій та взаємного розташування з іншими об’єктами простору.

Найбільше часу зайняло оптимізація генератора поверхів. На

початкових етапах створювались кімнати, які вже потім заміщувались на потрібні, але із-за такого підходу, виникало багато проблем з сусідством особливих кімнат та початкового положення персонажа. Саме тому, було запропоновано переробити генератор під поступове створення та додавання кімнат на вже згенерований шаблон.

Під час оптимізації коду, було створено багато допоміжних функцій для зменшення кількості повторюваності, але процес генерації поверхів все одно потребує створенню допоміжних функцій для виведення переборів сусідства кімнат тощо.

Колізії у грі працюють правильно. При взаємодії штучно згенерованого ворожого снаряду, гравець отримує шкоду, а самі снаряди гравця, наносять шкоду тестовому манекену.

Вивід ключових моментів генерації та змін параметрів персонажа, виводить у консоль розробника, що полегшує пошук проблем. Також, за допомогою механіки рушія, можна відслідковувати та впливати на змінні грі при її запуску та тестуванні.

3.5. Висновки до розділу 3

У цьому розділі ми:

- спланували проєкт, а саме: визначили головні параметри, концепцію, ключові механіки, процес гри, визначили етапи створення візуальної та програмної частини;
- підготували та організували робочій простір у середовищі програмування Godot Engine;
- створили файлову структуру та заповнили її необхідним матеріалом;
- попередньо налаштували проєкт, визначили рівні колізії, розміри екрану та параметри входження даних;
- реалізували графічну складову, створили персонажа, приклади кімнат, снаряди, анімації та користувацький інтерфейс;
- більш детальніше розібрали основні моменти створення графічної

складової та оглянули додаткові вкладки інтерфейсу, які знадобились;

- описали всі функції для реалізації: переміщення персонажа, його взаємодії з оточенням, випускання снарядів, створення шаблонів кімнат різних типів, оновлення користувацького інтерфейсу, процедурна генерація поверхів з кімнатами із особливими умовами.

ВИСНОВКИ

У ході написання даної бакалаврської роботи вдалося виконати основні завдання, а саме:

1. Проаналізовано історію зародження та розвитку жанру roguelike, досліджено ігри цього жанру. У роботі було успішно проаналізовано історію зародження та розвитку жанру Roguelike. Висвітлені ключові етапи його еволюції, починаючи з гри "Rogue" (1980), яка дала назву жанру та сформувала його основоположні принципи.

2. Були ретельно досліджені ключові характеристики обраного жанру, такі як процедурна генерація рівнів, перманентна смерть, покроковий режим, висока складність, елементи RPG та висока реіграбельність. Визначено, що саме ці принципи є фундаментом для створення захопливого та непередбачуваного ігрового досвіду, що знайшло відображення у авторській розробці даної кваліфікаційної.

3. Проаналізовані можливості рушія Godot Engine та визначено його переваги для розробки 2D-ігор. В ході дослідження виявлено, що Godot Engine є потужним та гнучким інструментом для розробки 2D-ігор завдяки відкритій ліцензії, архітектурі, орієнтованій на сцени та вузли, а також підтримці мови GDScript, яка забезпечує швидку розробку. Його кросплатформність та ефективний робочий процес були визнані оптимальними для реалізації даного проєкту.

4. Розглянуті можливості скриптової мови GDScript. Акцентована увага на тісній інтеграції GDScript з архітектурою Godot (система вузлів та сцен), що дозволяє ефективно взаємодіяти з елементами ігрового світу.

5. Завдання зі створення власної ігрової розробки було повністю виконано, що підтверджується детальною документацією процесу розробки, демонстрацією ключових алгоритмів (зокрема, генерації рівнів) та наявністю працездатного ігрового продукту, розміщеного у репозитарії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ariel Manzur, George Marques. Godot Engine Game Development. Pearson, 2021. 273 p.

2. Chris Bradfield. Godot Engine Game Development Projects. Packtx Publishing, 2018. 271 p.

3. Guiness World Record. guinnessworldrecords.com. URL: https://www.guinnessworldrecords.com/world-records/494516-first-open-world-survival-videogame.

4. Enter the gungeon official site. enterthegungeon.com. URL: https://enterthegungeon.com/.

5. GoDot Engine documentation. godotengine.org. URL: https://docs.godotengine.org.

6. Son Nguyen. Making a 2D Game with Godot 4 Engine. Tampere : Tampere University of Applied Sciences, 2024. 30 p.

7. Ангел М. В. Розробка 2D гри на платформі Godot : кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» : спец. 121 «Інженерія програмного забезпечення» / М. В. Ангел ; ЧНУ ім. Петра Могили. Миколаїв, 2024. 79 с.

8. Бойченко М. А. Гра MoonBallon [Електронний ресурс] URL: https://github.com/idejavyu/moonballon

9. Вікіпедія Brotato [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Brotato (дата звернення: 21.05.2025).

10. Вікіпедія NetHack [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/NetHack (дата звернення: 10.09.2023).

11. Вікіпедія Risk of Rain 2 [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/Risk_of_Rain_2 (дата звернення: 01.12.2023).

12. Вікіпедія Rogue (відеогра) [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/Rogue_(%D0%B2%D1%96%D0%B4%D0%B5%D0%BE%D0%B3%D1%80%D0%B0) (дата звернення: 14.02.2023).

13. Вікіпедія Roguelike [Електронний ресурс] URL:

https://uk.wikipedia.org/wiki/Roguelike (дата звернення: 30.01.2025).

14. Вікіпедія Rogue Legacy 2 [Електронний ресурс] URL: https://en.wikipedia.org/wiki/Rogue_Legacy_2 (дата звернення: 1.06.2025).

15. Вікіпедія SpurguX [Електронний ресурс] https://fi.wikipedia.org/wiki/SpurguX (дата звернення: 20.04.2024).

16. Вікіпедія The Binding of Isaac [Електронний ресурс] URL: https://uk.wikipedia.org/wiki/The_Binding_of_Isaac (дата звернення: 13.01.2024).

17. Возняк М. А. Дослідження можливостей рушія Godot для розробки гри комбінованих жанрів : спец. 122 Комп’ютерні науки / наук. кер. Т. О. Гришанович ; Волинський національний університет імені Лесі Українки. Луцьк , 2024. 59 с.

18. Дуров О. В. Ігровий застосунок в жанрі roguelike на рушії Unity : кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» : спец. 122 «Комп’ютерні науки» / О. В. Дуров ; ЧНУ ім. Петра Могили. Миколаїв, 2023. 66 с.

19. Ільїн, Г. А. Розроблення комп'ютерної гри жанру Roguelike : кваліфікаційна робота на здобуття ступеня вищої освіти «бакалавр» / Г. А. Ільїн ; наук. керівник канд. техн. наук, доц. Д. Л. Кирийчук. Херсон : ХНТУ, 2024. 61 с.

20. Кісь О. В. Модель процедурної генерації програмними засобами для спрощення розробки додатків : пояснювальна записка до кваліфікаційної роботи здобувача вищої освіти на другому (магістерському) рівні, спеціальність 123 Комп'ютерна інженерія / О. В. Кісь ; М-во освіти і науки України, Харків. нац. ун-т радіоелектроніки. Харків, 2022. 91 с.

21. Костомаха О. М. Розробка віртуальних ігрових симуляторів за допомогою рушія godot engine. Сучасні цифрові технологій та інноваційниі методики навчання: досвід, тенденції, перспективи. 2022. С. 16–18.

22. Кулеба В. О. Розробка 2D-гри в жанрі roguelike з динамічною генерацією рівнів засобами рушія Unity : кваліфікаційна робота на здобуття

освітнього ступеня «бакалавр» : спец. 122 «Комп’ютерні науки» / В. О. Кулеба ; ЧНУ ім. Петра Могили. Миколаїв, 2024. 73 с.

23. Мартинюк С. В. Godot engine – інструмент для підготовки фахівців у сфері інженерії ігрових проєктів. Сучасні цифрові технологій та інноваційниі методики навчання: досвід, тенденції, перспективи. 2024. С. 155–159.

24. Полухін Ю. С. Ігровий застосунок в жанрі Roguelike. Розробка ігрового середовища : кваліфікаційна робота на здобуття освітнього ступеня «бакалавр» : спец. 121 «Інженерія програмного забезпечення» / Ю. С. Полухін ; ЧНУ ім. Петра Могили. Миколаїв, 2023. 63 с.

25. Прудіус В. Ю. Ігровий програмний застосунок у жанрі RPG з елементами економічної стратегії та roguelike. Економічні механіки, механіки на рівні історії : пояснювальна записка до кваліфікаційної роботи здобувача вищої освіти на першому (бакалаврському) рівні, спеціальність 121 – Інженерія програмного забезпечення / В. Ю. Прудіус ; М-во освіти і науки України, Харків. нац. ун-т радіоелектроніки. Харків, 2024. 115 с.

26. Скороход М. М. Розробка комп'ютерної гри «Forgotten Island: Hero Artifact» / М. М. Скороход // Радіоелектроніка та молодь у XXI столітті : матеріали 29-го Міжнар. молодіж. форуму, 16–19 квітня 2025 р. Харків : ХНУРЕ, 2025. Т. 5. С. 55–56.